Michael Würsch

Dissertation

A QUERY FRAMEWORK FOR SOFTWARE EVOLUTION DATA

Advisors

Prof. Dr. Harald C. Gall
University of Zurich

Prof. Dr. Serge Demeyer
University of Antwerp

University of Zurich[UZH]

Hawkshaw

A Query Framework for Software Evolution Data

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology of the University of Zurich

for the degree of
Doctor of Science

by
Michael Würsch
from Zurich, ZH, Switzerland

Accepted on the recommendation of
Prof. Dr. Harald C. Gall
Prof. Dr. Serge Demeyer

2012

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, September 2012

Head of the Ph.D. committee for informatics:
Prof. Abraham Bernstein, PhD.

Acknowledgements

A much wiser man than me once said that feeling gratitude and not expressing it is like wrapping a present and not giving it. I would therefore like to embrace the opportunity and thank the many people that have supported me in writing this thesis.

First and foremost, I thank my advisor, Harald Gall, for his guidance and support during my Ph.D. studies. He taught me to strive for excellence and always encouraged me to pursue my own ideas. No matter how busy his schedule was, he always managed to spare some time for me when I was seeking advice. And we have been together to many exotic places—from deserts to jungles—where we had some great adventures and also a lot of fun.

Special thanks go to Serge Demeyer for flying all the way to Zurich to be part of my Ph.D. committee as an external examiner. I want to thank him for dedicating his precious time to evaluate my work and for providing very valuable feedback—not only during the final phase of my Ph.D. thesis, but also while staying in our group during his sabbatical. It has been very inspiring to witness his passion for research and to get carried away by his enthusiasm for finding out what makes software development work (and what does not).

I could not have possibly made it through all the ups and downs that are part of the academic life without my fellow researchers and doctoral students from the SEAL lab at the University of Zurich. It has been a pleasure and a privilege to work with you all. Thanks to Emanuel Giger, whom I met on my way to the office on my very first day of work and who has accompanied me to various locations around the world ever since (and even to some virtual places as well). I acknowledge Matthias Hert for his great sense of humor and for spending countless hours on finding out how I broke the Semantic Web (again) and what could be done to fix

it. I thank Gerald Reif and Martin Pinzger for their encouragement and advice, as well as for the great time we spent together. I am also grateful to Beat Fluri for taking me under his wings when I started my Ph.D. and for all the fun we had while writing papers together. Thanks to Giacomo Ghezzi for all the entertaining discussions we had on our latest achievements in cooking and barbecuing, but also for his work as a co-author of a number of papers that eventually found their way into this thesis. I would also like to thank Amancio Bouza for all the laughs that we had together and Patrick Knab for the enjoyable discussions we had in the past. I appreciate the help of Sebastian Müller who stepped in and relieved me from my teaching duties when I had to focus more on research, as well as the help of Martin Brandtner in keeping all our infrastructure alive. I see a bright future ahead for both of them and I am proud to be part of it for a little bit longer. Thanks to Sandro Boccuzzo, Yvonne Engeler, Thomas Fritz, Yi Guo, Chaman Wijesiriwardana, Yu Zhou, and Jonas Zuberbühler.

Thanks to the current and past members of the DDIS group, in particular Avi Bernstein, Adrian Bachmann, Lorenz Fischer, Christoph Kiefer, Patrick Minder, Katharina Reinecke, Thomas Scharrenbach, Jonas Tappolet, Peter Vorburger, Cathrin Weiss, and especially to Esther Kaufmann for providing me with the source code of her Ginseng tool.

Many thanks to my dear friends Kristina and Reto with their lovely daughters Anina and Eline, to Cornelia and Andreas, and to Franziska and Daniel. Their friendly faces cheered me up whenever I escaped the ivory tower for a while.

Very special thanks go to my parents, who have believed in me since day one. For my entire life, they encouraged me to pursue my own dreams and I could always count on their unconditional support no matter the challenge. I thank my sister for being there when it counted.

Finally, my deepest and most sincere appreciation goes to my significant other, Claudine, and my beloved daughter Emily for all their love, patience, and understanding throughout these years. My love and gratitude extend beyond words and reasons.

Michael Würsch
Männedorf, September 2012

To Claudine and Emily

Abstract

The feature list of modern integrated development environments is steadily growing and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, development environments often still cannot directly answer the questions that arise during program maintenance tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using a quasi-natural language interface that requires almost zero learning effort. Our approach is tightly woven into the Eclipse development environment and allows developers to answer questions related to source code, development history, or bug and issue management. For that, we model data extracted from various software repositories by means of ontologies, store them in a knowledge base of software evolution facts, and use knowledge processing techniques from the Semantic Web to query the knowledge base. Our approach was evaluated in a user study with 35 subjects, who had to solve various software evolution tasks for an industrial-scale, open-source software system. The results of our user study showed that our query interface can outperform classical software engineering tools in terms of correctness, while yielding significant time savings to its users and greatly advancing the state of the art in terms of usability and learnability.

Zusammenfassung

Moderne integrierte Entwicklungsumgebungen werden von ihren Herstellern mit jeder neuen Version um zahlreiche Funktionalitäten erweitert. Mit wachsender Funktionsvielfalt leiden Zugänglichkeit und Benutzbarkeit, sodass es speziell für unerfahrene Entwickler zunehmend schwieriger wird all die Entwicklungswerkzeuge zu meistern, die ihnen an die Hand gegeben werden. Trotz der bemerkenswerten Eigenschaften der Entwicklungsumgebungen ist es somit oft alles Andere als trivial jene Fragen zu beantworten, die sich Programmierern während ihrer täglichen Arbeit stellen. Ein Teil des Problems liegt darin begründet, dass es zwar oft möglich ist ein Problem zu benennen, sich jedoch das Finden einer Lösungsstrategie schwierig gestaltet, da letztere sich dann auch auf die zur Verfügung stehende Funktionalität abbilden lassen muss. Entwickler müssen folglich das Problem zuerst in solche Teilprobleme zerlegen, die sich mit den vorhandenen Werkzeugen auch beantworten lassen. Im Anschluss ist es dann erforderlich die Teilresultate – oft erst nach vorgängiger manueller Analyse – wieder zu einem Ganzen zusammenzusetzen.

Bestehende Ansätze um diese Arbeit zu erleichtern haben oft den Nachteil, dass entweder nur eine beschränkte, fix vordefinierte Menge von Abfragen unterstützt wird, oder dass stattdessen von Entwicklern erwartet wird, dass diese extra eine formale Abfragesprache nur zu diesem einen Zweck erlernen. Wir präsentieren hingegen ein Rahmenwerk um Information über ein in Entwicklung befindliches Computerprogramm mit Hilfe einer quasi-natürlichsprachigen Benutzerschnittstelle abzufragen, welche ohne nennenswerten Lernaufwand verwendet werden kann. Unser Ansatz ist vollständig in die Eclipse Entwicklungsumgebung integriert und erlaubt Entwicklern Daten betreffend Programmcode, dessen Entwicklungsgeschichte, sowie dessen Fehlerhistorie abzufragen.

Zu diesem Zweck modellieren wir die Daten, welche wir aus verschiedenen Programmrepositorien extrahieren, mit Hilfe eines Metamodells basierend auf einer Ontologie. Das Modell speichern wir dann in eine Wissensdatenbank und verwenden Techniken zur Wissensverarbeitung, welche wir dem Semantischen Netz entlehnen, um das Modell abzufragen.

Unser Ansatz wurde im Rahmen einer Benutzerstudie mit 35 Personen evaluiert. Die Teilnehmer mussten verschiedene Aufgaben betreffend der Wartung und Evolution eines industriellen, quelloffenen Computersystems lösen. Die Resultate der Benutzerstudie zeigen, dass unsere Benutzerschnittstelle statistisch signifikante Einsparungen bezüglich der Zeit ermöglicht, welche für das Lösen der Aufgaben aufgewendet werden musste. Einen sogar noch grösseren Vorsprung gegenüber bestehenden Ansätzen konnten wir im Hinblick auf Benutzbarkeit und Erlernbarkeit unseres Ansatzes messen.

Contents

List of Figures

List of Tables

List of Listings

A prudent question is one-half of wisdom.

—Francis Bacon

(English Lawyer and Philosopher. 1561-1626)

Question everything. Learn something. Answer nothing.

—Euripides

(Greek playwright, c. 480-406 BC)

The power to question is the basis of all human progress.

—Indira Gandhi

(Indian Politician and Prime Minister. 1917-1984)

1 Synopsis

Ever since the software crisis in the 1960s, software engineers have sought to come up with methods, processes, and tools to improve software construction. Despite decades of research and industrial practice, it is still difficult and therefore costly to build and maintain reliable software that is easy to adapt to the constantly evolving environment it is embedded in.

A surprising observation is that costs arise in large parts for maintenance of a software system, rather than during its conception and implementation phases. Some estimates even claim that 85% to 90% percent of IS budgets goes to legacy system operation and maintenance [Bro95, Erl00]. One problem is the gestalt of software which is beyond comparison with any other phenomena encountered in the history of mankind: legacy systems are typically complex and large—ranging from hundred thousands to dozens of millions lines of code—but at the same time intangible, with no physical shape or size [BE96]. Software engineers have to maintain complex mental models to grasp their problems at hand when given the task to change or repair existing code [LVD06]. To make matters worse, Lehman attributes the fundamental difficulties to the process of software development itself, which is *"managed and implemented by people; thus in the long term [its behavior should] be expected to be unpredictable, dependent on the judgments, whims, and actions of [people]."* [Leh79]

Lehman was also among the researchers who coined the term software evolution in the seventies and early eighties. In measuring certain aspects of large

industrial programs of that time, he and his colleagues found evidence for multiple regularities or laws in the software life-cycle. Lehman recognized that those laws were closer to biological laws or the ones of modern physics than to those formulated in other areas subject to human influence, such as economics and sociology. Thus they were given the name *"Laws of Software Evolution"* [Leh80]. Eight laws have been discovered up to now, but the following three summarize the main causes for continuous maintenance need sufficiently well:

- **Continuing Change.** Software systems must be continually adapted or they become progressively less satisfactory.
- **Increasing Complexity.** As a software system evolves its complexity increases unless work is done to maintain or reduce it.
- **Continuing Growth.** The functional content of software systems must be continually increased to maintain user satisfaction over their lifetime.

With the advent of the first software configuration system (SCM) [Roc75], project trackers, and modern integrated software development environments (IDEs), developers were given powerful tools to respond to some of the challenges of software maintenance. SCMs permit developers to coordinate better their efforts to produce large software systems by providing mechanisms for concurrent modification of software artifacts. Project trackers allow engineers to record bugs and issues, as well as to manage the process of resolving them. Both SCMs and project trackers are repositories for software artifacts and provide a valuable aid for managing changes and sustaining growth. Besides basic editing capabilities, IDEs offer views of source code on different levels of abstraction and facilitate searching in large code bases; they help to harness complexity.

A number of researchers have recognized that such software (artifact) repositories accumulate large amounts of historical data that can give deep insight into the evolution of a software development project. Retrospective analysis of this data allows for both finding latent problems in programs and for predicting future development of a software system. Among the latent problems are, for example, logical couplings between files [GHJ98, FPG03b] or critical bugs [DLP07]. Prediction approaches range from identifying components that are likely to change again in the future [GDL04] to the prediction of bugs [DLR12]. Attracted by this flourishing field of research, a vivid community of researchers has formed over the

last three decades—a community that gathers at an increasing number of venues, for example, at the *International Conference on Software Maintenance*,[1] the *European Conference on Software Maintenance and Reengineering*,[2] or the *Working Conference on Mining Software Repositories*.[3]

1.1 Motivation and Thesis Statement

The rich software evolution data contained in software repositories may yield many interesting opportunities to researchers, but at the same time our claim is that the full potential of the stored knowledge cannot yet be realized by software developers in practice. In particular, we see two main obstacles that hamper the accessibility of the data: the first one is that neither classical version control systems, nor bug and issue trackers provide sufficient support for a detailed analysis of software evolution aspects [FPG03b]. Many relations among the data they store are implicit, the repositories are *information silos* in the sense that they lack integration between them, and they are hardly queryable in a uniform way (if at all). For example, popular SCMs such as CVS[4] and Apache Subversion[5] (SVN) do not expose any querying functionality, *i.e.*, no (graphical) search interface, repository query language, or search API. Instead, developers are limited to to splicing together the outputs from multiple command line tools, followed by a manual evaluation and interpretation of the results. Commonly used bug and issue trackers, for instance, Bugzilla,[6] do incorporate a Web-based user interface with input masks for querying. However, the query features are hardly more sophisticated than simple keyword-based search and they cannot incorporate any other artifacts, stored in other repositories. For example, understanding the rationale behind a recent change in the program code therefore may first require tedious browsing of change logs provided by a version control system in order to find textual references to reported bugs; then switching to the Web interface of an issue tracker for reading through the corresponding bug descriptions and associated customer-developer discussion threads. When no rigid change process

[1] `http://conferences.computer.org/icsm/`
[2] `http://csmr.eu/`
[3] `http://msrconf.org`
[4] `http://www.nongnu.org/cvs/`
[5] `http://subversion.apache.org/`
[6] `http://www.bugzilla.org/`

is followed and bug references are lacking, it may even be impossible to recover such links.

The second major obstacle lies in the considerable amount of time and learning effort that software engineers have to devote themselves to, in order to achieve mastery in the advanced features of IDEs and the many different software engineering support tools that are used in practice nowadays. The difficulties are partially grounded in the absence of one consistent, common usability concept across the tools and repositories or—even better—a single point of access for common information needs of software engineers. Hence, when developers use their IDE, or when they access different software repositories to solve maintenance tasks, their problem is often not that they do not know *what* to query, but rather *how* to formulate their particular question in a way so that it can be answered with the tools available. In other words, as argued by De Alwis and Murphy in [dAM08], developers often have conceptual queries in mind, but they first need to map these conceptual queries to one or several concrete queries. Establishing such a mapping can already be difficult to achieve on its own—it is even harder if partial results obtained from different tools require composition. The extent of these challenges becomes evident, for example, when project managers staff their best people in the product team, while keeping the junior developers in the maintenance team. The latter then need to understand the code and design when they were not part of the team that took the decision in the first place, while it is also expected of them that they quickly gain proficiency in using the many software engineering tools that their development teams rely on.

In recent years, research has gained good level of understanding of what kind of conceptual queries arise when software engineers are maintaining programs. For example, LaToza *et al.* [LVD06] performed an exploratory study with developers at Microsoft and found that most information needs were related to program comprehension, refocusing after task switching, program modularity, links between software artifacts, and team awareness. Ko *et al.* performed a similar study and reported the different maintenance and development activities during which questions arise: writing code, submitting a change, triaging bugs, reproducing failures, understanding program behavior, reasoning about design, and again, becoming aware of the work that other team members are doing [KDV07]. Another catalogue of mostly source-code-related questions was compiled by Sillito *et al.* The catalogue contains common questions, such as *"Where is this method*

called?" and *"What are the subclasses of this class?"* Each question was classified by one of the following categories: finding initial focus points and building on them, understanding a subgraph of information, and questions over groups of subgraphs [SMDV06, SMV08]. A conclusion drawn by several of these researchers was that, in principle, many of the reported information needs were already supported by commonly available software engineering tools. But in practice, software engineers faced substantial challenges nonetheless. One observation was that maintainers had to rely heavily on implicit knowledge [LVD06]. This becomes an issue when they are lacking the knowledge because they yet have to graduate to more seasoned software engineers or when they have to take over tasks from other developers no longer available for consultancy. Other difficulties were encountered whenever it came to mapping questions onto programming tools, and using the results from those tools to answer the intended question [SMDV06].

One might think that, once the major problems and information needs of developers are identified, it is only a matter of time until research comes up with a solution to tackle and answer them, respectively. However, Brooks postulated already more than 25 years ago that,

> *"There is no single development [in software engineering], in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity."* [Bro87]

He attributes several aspects of the mentioned obstacles to the intrinsic complexity of the software development process, which is thus not amenable to any *"silver bullets;"* there is simply no one-size-fits-all solution available and software engineers will have to live with a heterogenous tool landscape to a certain extent. Still there is room for improvement and Brooke, in that respect, also claimed that,

> *"Perhaps the biggest gain yet to be realized from programming environments is the use of integrated database systems to keep track of the myriad details that must be recalled accurately by the individual programmer and kept current for a group of collaborators on a single system."*

Researchers have picked up the idea of database systems integrated into an IDE and came up with formal query languages, for instance, CodeQuest [HVdM06] or JQuery [JV03]. These approaches offer a database of queryable facts, so that developers can formulate more sophisticated queries about a limited set of software

artifacts. Most of these approaches only support source code queries, *i.e.*, no other software evolution data can be queried. The formal languages are usually based on SQL- or Prolog-like syntax and developers need to familiarize themselves closely with the underlying data representation in order to be capable of formulating valid queries. Hence, effectively using such languages requires considerable learning effort. However, Hallet *et al.* recognized that even when developers go the extra mile to learn such a language, mistakes in query formulation often remain difficult to spot for them. In consequence, when a query returns without results, it can be non-obvious whether this is because an answer was not available in the system being queried or because the query construction was improper. Even if a result is returned, the answer may still not be an accurate response to the intended question [HSP07]. All these deficiencies are a potential harm to the confidence of users in their query systems and may be one of the reasons why formal query languages for software artifacts have not found currency in industry yet, even two decades after their first conception.

According to Chowdhury, *"the most comfortable way for a user to express an information need is as a natural language statement."* [Cho04]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [Hen94]. However, while natural language clearly provides intuitive means to pose questions, *"queries expressed in free natural language are [also] very sensitive to errors of composition (e.g., misspellings, ungrammaticalities) or processing (at the lexical, syntactic, or semantic level)."* [HSP07].

The method of *Conceptual Authoring* promoted by Hallet *et al.* in [HSP07] strikes a balance between free natural language and more structured queries. For composing queries, all editing operations are defined directly on an underlying logical representation, an ontology. However, users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology. Since users still build the logical representation directly, Conceptual Authoring depends entirely on language *generation* technology and avoids the pitfalls of language *interpretation* completely.

Our premise is therefore that Conceptual Authoring is a valuable approach for answering common conceptual queries of developers and we hence state our thesis as follows:

Thesis Statement: Software engineers will benefit from a quasi-natural language interface that uses Conceptual Authoring to enable a wide range of queries against an integrated knowledge base containing various facts about the evolution of the software system they are working on.

1.2 Research Questions

The overall goal of this thesis is to provide software engineers with a convenient and intuitive query interface which allows for answering common information needs that arise during software development and maintenance. We aim at reducing the cognitive burden placed on developers that goes hand in hand with translating conceptual queries to a concrete ones in the context of their daily programming activities.

With respect to the overall goal and our thesis statement, we raise three research questions, which are detailed in the following. Figure 1.1 depicts how they relate to the remaining chapters of the thesis.

RQ1: Can we provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactical complexity?

The first research question addresses the feasibility of the envisioned approach and can be decomposed into three important aspects: knowledge representation, knowledge integration, and knowledge accessibility.

Accessibility is related to the ease of retrieving desired information from a knowledge base by our end-users, the software engineers. It is directly impacted by the human-computer interface we incorporate into our approach. As argued in Section 1.1, for effectively using formal query languages, one needs to be deeply

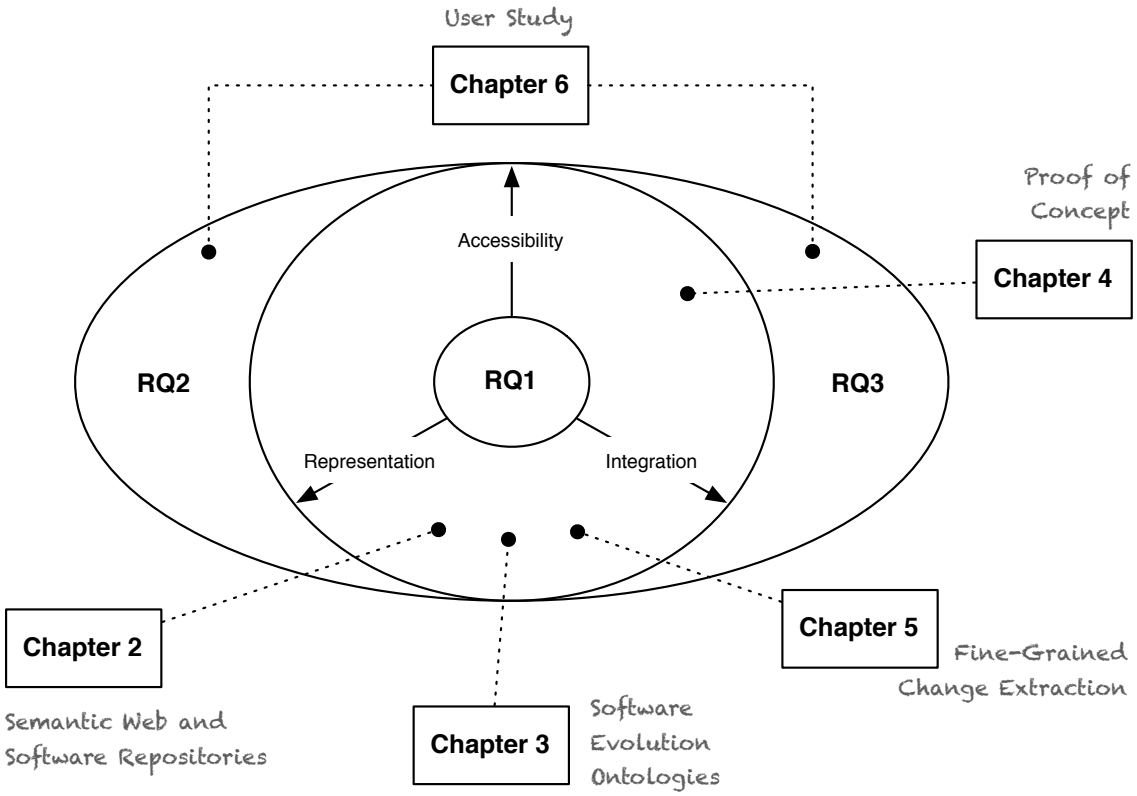


Figure 1.1: Relation between the Research Questions and the Chapters of the Thesis

familiar with both syntax and schema of the data being queried. On the other hand, many traditional menu-driven or command-line-based user interfaces are either inflexible (*i.e.*, limited to one particular kind of information need) or quickly can become very complex in use. In the latter case it is often non-obvious how a conceptual question can be answered. To succeed, good information retrieval strategies are needed, which again implies a fair amount of experience with the tools under usage. We therefore explore in this thesis how we can implement a guided-input natural language interface that, in contrast, guides developers closely in directly formulating conceptual queries and assumes basically no prior knowledge of the underlying approach.

Representation is concerned with an adequate formalization of the knowledge, so that it becomes queryable with our interface. Classical relational database schemas lack the explicit semantics to support our approach. In the relational paradigm, we would therefore be required to hard-wire a substantial amount of

additional knowledge into the query component of our envisioned framework, making it less extensible with respect to the incorporation of other software evolution data. The Semantic Web offers a valuable alternative in that it provides means to formally describe the semantics of software repository data. Hence, in the course of this thesis, we will investigate which concepts should be modeled in order to enable the queries that correspond to the most common information needs of developers.

Integration takes into account that most software repositories act as information silos, with no links between the artifacts or the associated meta-data that each of them stores. Many important relations are therefore only of an implicit nature. Making these relations explicit and therefore queryable is crucial for the acceptance of the query approach described in this thesis, as it would relieve developers from the tedious task of composing information fragments obtained from multiple queries performed with different tools. A number of algorithms and heuristics for integration already exist, *e.g.*, to link bugs and issues to changes. In the context of this thesis, we make use of them and also contribute additional ones.

RQ2: When developers use a quasi-natural language interface to satisfy their information needs, are they able to successfully formulate and enter common developer questions, and can we achieve an advancement over the state of the art in terms of time efficiency in retrieving the answers or in their correctness?

The second research question depends on the first one being answered positively. It addresses the actual, measurable benefit that our approach brings to developers in terms of an increase in their efficiency when solving common software evolution and maintenance tasks. By "increase in efficiency," we mean either a reduction of time for retrieving answers to common developer questions or a higher quality of the answers. We answer RQ2 based on the empirical results obtained from a user study. The answer particularly depends on the rejection or acceptance of the corresponding hypotheses presented in Table 6.2 on page 173. In the user study, we assigned a set of tasks to one group of subjects that were allowed to use our query interface, as well as to a second group that could only rely on a baseline of traditional tools. We derived the tasks from the existing

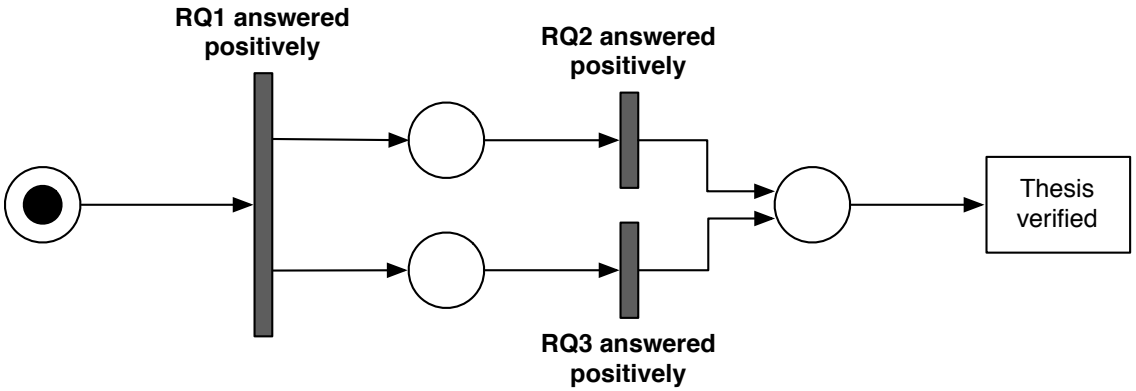


Figure 1.2: Thesis Verification State in Relation to the Answers of RQ1 – RQ3

catalogues of common developer questions that were already briefly mentioned in Section 1.1. The efficiency of both groups in solving the tasks was then compared and we considered the answer to RQ2 positive if we could observe for our interface, in comparison to the baseline, either time savings, improvements in terms of correctness, or both.

> **RQ3:** Is a quasi-natural language interface well-accepted by the users or do they prefer traditional means to access data about their software systems, *i.e.*, those tools that are already provided by common IDEs, issue trackers, version control systems, and Web search engines?

The final research question again assumes that RQ1 was answered positively. When given the choice, users in general (and software engineers in particular) do not necessarily use the most powerful tools available but rather choose whatever they feel most convenient in working with. In raising the third research question, we acknowledge this fact and take into account overall system satisfaction, as well as usability and learnability of our approach. In the course of the user study, we also incorporated the System Usability Scale (SUS) [Bro96] to measure satisfaction. The SUS further provides subscales for usability and learnability. The answer to RQ3 was considered positive, if the subjects of the user study reported higher SUS (sub-)scores for our approach than for the baseline.

Figure 1.2 shows a Petri net schematic for the verification state of our thesis. A

positive answer to RQ1 is a prerequisite for answering RQ2 and RQ3, as well as for verifying the thesis itself; a negative answer would have disproven our thesis immediately. Out of the two other research questions, RQ2 and RQ3, at least one needed to be answered positively in order to verify our thesis. In other words, we considered our research successful if (1) a proof of concept of our approach was achieved and (2) the results of the user study showed that subjects can solve the tasks with our interface more efficiently than with the baseline, or if the subjects experienced a higher system satisfaction with our approach.

1.3 Hawkshaw in a Nutshell

In the context of this thesis, we devised a framework that allows software engineers to use guided-input natural language strongly resembling plain English to query for information about the evolution of a software system. This includes queries related to source code, development history, as well as to bug and issue management. The major benefit of such a query interface is that it requires almost zero learning effort from users before they are able to successfully enter a query, *i.e.*, no prior knowledge of query syntax or underlying data schema is assumed.

Our approach is called HAWKSHAW. The name that stands for both, the conceptual framework and its proof-of-concept implementation.[7] The latter is provided in terms of a tool that extends the Eclipse IDE with additional querying functionality. The core components of the framework are the guided-input natural language interface, ontologies that provide a queryable formalization of software evolution knowledge, and a set of fact extraction algorithms used to populate the ontologies with instances of real software systems. The guided-input natural language interface was inspired by the Ginseng approach of Bernstein *et al.* [BKKK06].

Figure 1.3 shows a screenshot of our approach in action. The query dialog is fully integrated into the *Search* menu of Eclipse, but can also be brought up anytime by pressing a shortcut. In the example, a user has already started to compose a query: three words have been typed in so far (*i.e.*, *"What"*, *"Method"*, and *"invokes"*) and the drop-down menu presents the full list of method names that can be entered

[7] We named our framework after *Hawkshaw the Detective*, a comic strip popular in the first half of the 20th century. *Hawkshaw* meant a detective in the slang of that time. One important common ability of detectives is to ask the right questions in the right context. Our framework's aim is to support "software detectives" in investigations into their programs.

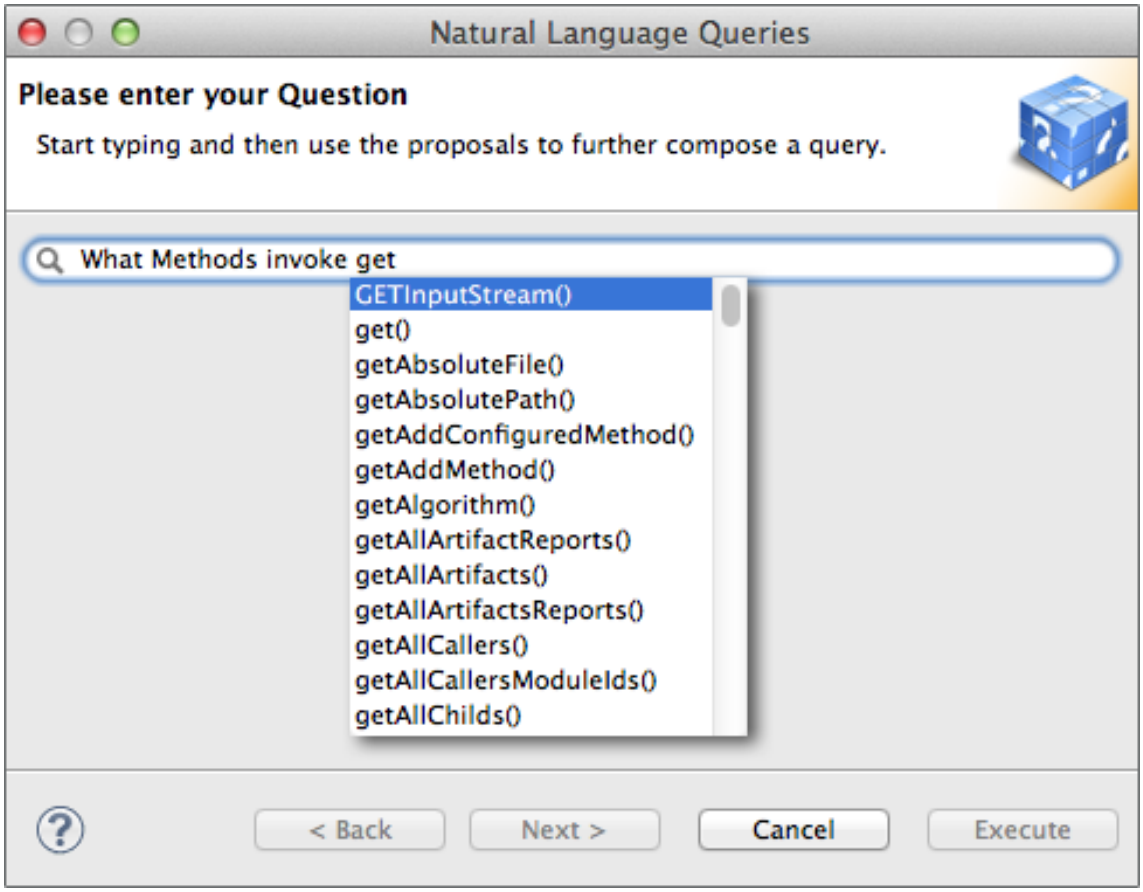


Figure 1.3: Hawkshaw in Action

to complete the query. Only meaningful words are proposed: exclusively those verbs are shown that semantically fit the grammatical subject, and only objects matching the verb can be selected. However, users can type freely, as long as the entered characters match at least one of the proposed words. Therefore, our approach guides developers closely in formulating their information needs in a way such that the resulting query is processable by the query system. Developers quickly receive feedback about the range of possible queries through the proposals we show them and we can prevent them from entering invalid questions, not understandable by our query system. The complete question is translated into a formal query language statement, which is then executed against a knowledge base of software evolution facts.

HAWKSHAW is not limited to answering questions related to a source code snapshot. It also incorporates a wide range of concepts from SCMs or bug and

issue trackers, and relies on sophisticated algorithms to establish links between the facts extracted from the different software repositories. Hence, it is possible to ask a much broader range of questions. Examples for such questions are:

- What bugs affected this method?
- What classes were [recently] changed by my co-worker?
- What package is most error-prone?

In the remaining chapters of the thesis, we show many additional examples for questions that can be entered and then answered with our approach. We further provide a detailed description of the underlying fact extraction and query composition algorithms, as well as of the information model used. Each chapter represents a scientific publication that contributes to the foundation of the thesis. A summary of the foundation is given in the next section.

1.4 Foundation of the Thesis

In the following, we briefly summarize the main contributions and findings from five selected articles. All five articles were published in internationally renowned, peer-reviewed venues. The venues include one workshop, one international conference, and three different journals in the field of software engineering. Together, the five articles form the foundation of this thesis. The full papers are presented in Chapters 2–6.

Chapter 2: *How Semantic Web Technology could influence Software Repositories.* The chapter's overall contribution to this thesis is two-fold: first and foremost it motivates our design decision to build the integrated knowledge base of software evolution facts in compliance with the principles of the Semantic Web, instead of relying on classical relational database technology. Second, we also sketch our long-term vision for how software repositories should expose the data they store, and how better integration between different repositories can be achieved. While the expected impact is much broader, our approach to querying software evolution data would greatly benefit from our vision being materialized. Before we further elaborate on the

particular benefits, we briefly summarize our main arguments to support the incorporation of the Semantic Web into software repositories.

We argue that current software repositories are rigid and inflexible monoliths, with lack of cross-repository integration and very limited analysis and query support. Our proposal is therefore to build on Semantic Web Technology, such as Linked Data and ontologies, when constructing the next generation of software repositories. In our vision, software repositories should ultimately blend into a queryable global information space of interlinked data about all kinds of software engineering artifacts. Our arguments are in line with the proposal published by Tappolet in [Tap08], but we additionally describe four typical problem scenarios in the context of software analysis and software repository infrastructure. These scenarios contrast key challenges that cannot be easily overcome by traditional solutions with the relief that the Semantic Web brings in that respect. In particular, we identified the following general areas for improvements:

- **A Shared Vocabulary.** One of the critical design aspects when building a knowledge base is to define a meta-model that formalizes the knowledge in an adequate level of detail. Also, in order to share data among different tools, they need to understand the same vocabulary. Existing meta-models, for instance, the FAMIX meta-model [TDD00] or the Dagstuhl Middle Meta-Model (DMM) [LTP04], rarely follow a reasonably formal specification, but rather are their schemata defined in terms of natural language descriptions, sometimes augmented by models produced with visual modeling languages (usually UML class diagrams). Some of them also leave the serialization syntax open to implementors, who then decide for a non-semantics-preserving, custom format to transmit or store the model elements. In practice, the lack of specification and formalization results in many different, mutually incompatible implementations of the same concepts, thus wasting many potential synergies.

  With the Web Ontology Language (OWL) [De04] and the Resource Description Framework (RDF) [Ke04], the Semantic Web provides tools explicitly designed to formalize and share knowledge. OWL allows for describing concepts and their relationships with a notation that is

based on description logics, whereas RDF offers a common format for serialization.

- **Linked Data.** To interlink repositories on the Web, a flexible data model is needed that allows to expose references to data stored within a repository to the outside. Existing integration approaches do not allow exposition of artifacts on the Web, but rather store all the data in one monolithic relational database (*e.g.*, [GFP09, BOL09]).

  RDF is based on the concept that all statements about resources are made in terms of triples of uniform resource identifiers (URIs), which is simple yet powerful. Thus, RDF can be used to build an internet-scale graph of information since it helps to link and query data that is stored in different locations. Queries on such a graph are also possible thanks to the query language SPARQL [Pe08].

- **Traceability of Results.** The origin of scientific results should be traceable and independently replicable. It has to be made clear what data has been analyzed, *i.e.*, what meta-data needs to be supplied along with the results. But also the analysis itself needs to be described in similar detail. Unfortunately, both is still rather an exception than the rule, and only recent initiatives such as the PROMISE Software Engineering Repository [SSM05] to a certain extent alleviate the poor traceability and replicability of the scientific results reported by researchers in our field.

  RDF statements and URIs describe data under analysis accurately and authors can make the SPARQL queries public after they have used them in their analysis. Since the results are represented again as RDF statements, they can directly be processed in other tools for further investigations.

- **Relations.** There is no consistent way to get the meaning of a relation in relational databases. In fact, a query can join tables by any columns which match by data type—without any check on the semantics. While humans can often guess the meaning of a relation, computers cannot. With traditional means of technology, it is therefore necessary to encode a significant amount of implicit knowledge into our applications to make use of the data.

> The SPARQL query language allows one to query explicitly for relations among resources. These kind of queries are impossible in the relational and in the object-oriented paradigm, unless relationships are explicitly mapped to tables or, in the case of object-orientation, modeled as association classes. Given the high importance of relationships between artifacts and stakeholders in software engineering, it would be preferable to model them as first class objects—which is exactly what the Semantic Web does.

To take advantage of the Semantic Web like envisioned in the scenarios we came up with, we identified two major challenges and defined a research plan accordingly. These challenges are:

- **Challenge 1:** Formalize a common vocabulary to describe software artifacts and their relationships in terms of an ontology.
- **Challenge 2:** Devise strategies to generate URIs for artifacts of software projects.

Our solution for the first challenge is explained in Chapter 3, where we present the SEON ontologies, our formalization effort for different facets of the evolution of a software system. We tackled Challenge 2 in Chapter 4 for a subset of the artifacts stored in common software repositories. For the general case, we consider the problem still unresolved.

So far, we explained our vision of software repositories that blend into one global information space, and we initially claimed that this would significantly leverage the overall approach presented in this thesis, *i.e.*, a quasi-natural language interface that uses Conceptual Authoring to support software engineers in answering common questions associated to software evolution tasks. Our claim is based on the idea that, when SCMs and issue trackers would expose a SPARQL end-point for remote querying in the future, then the end-points could be registered with our query interface. Our framework would then perform federated SPARQL queries against the different end-points, without the need of additional import and preprocessing steps.[8] The theoretical foundations are currently being laid in research con-

[8] Alternatively, centralized development servers, for instance, the IBM Rational Team Concert, would render query federation unnecessary.

cerned with federated SPARQL querying (*e.g.*, in [QL08,BB10]). However, for the proof-of-concept implementation of our approach, we instead replicate the data from SCMs and issue trackers in a single knowledge base stored in a local triple store. The replication process is explained in Chapter 4, with a second variant being discussed in Chapter 6.

Chapter 3: *A Pyramid of Ontologies for Software Evolution and its Applications.* The outcome of the work is SEON. The acronym stands for *Software Evolution ONtologies* and these ontologies constitute our answer to the knowledge representation aspect of the first research question of this thesis, **RQ1**. SEON is organized as an ontology pyramid, with four different layers of abstraction, and it describes a broad range of software evolution concepts related to different stakeholders, their activities, involved software artifacts, and the relations among all of them. Concepts are modeled by OWL classes. Instances of classes are OWL individuals. OWL datatype properties represent attributes, and OWL object properties the relations between concepts. The first ones link individuals to data values, whereas the latter ones link individuals to individuals. Next, we briefly describe the essence of each layer of SEON:

- **General Concepts.** The pyramidion defines concepts omnipresent in software evolution, for instance, files or stakeholders. These concepts are independent of any particular information subdomain. Similarly, the domain-independent object properties are fundamental to many applications, as relations between "things" are paramount for most analyses in software evolution. They include generic relations, such as authorship and dependencies.
- **Domain-Spanning Concepts.** The second-highest layer defines concepts that are less abstract than the general ones. The layer is key to the HAWKSHAW approach since it describes how different aspects of the evolution of a software system integrate with each other. The domain-spanning concepts formalize knowledge that spans a limited number of subdomains. Notable in that respect is the ontology for fine-grained source code changes, which describes program modifications not only on a file level but also down to the statement level. It is based on the CHANGEDISTILLER meta-model of change types, which is described

in Chapter 5. The fine-grained changes bridge the gap between our source code model and the change history provided by SCMs; they are enablers for queries related to the history of single source code entities in HAWKSHAW.

- **Domain-Specific Concepts.** The third layer is divided into different domains that correspond to important facets of the software evolution process, *e.g.*, issue and version management. It includes a taxonomy for source code artifacts encountered in object-oriented programming. While the concepts defined in this layer are specific to a domain, such as object-oriented source code or bug tracking in general, they are independent of technology, vendor, and version. Each domain captures the commonalities shared among the many different issue trackers, object-oriented programming languages, or version control systems. Basically every question that can be answered with HAWKSHAW touches one or several concepts defined in this layer.
- **System-Specific Concepts.** Whereas the third layer describes domain-specific concepts that apply to families of systems, the bottom layer defines system-specific concepts. It extends the knowledge of the upper layers by concepts unique to certain programming languages, vendors, versions, or specific tool implementations.

Ontologies are described in terms of triples of subject, predicate, and object. This structure strongly resembles how humans talk about things and can be easily transformed into natural language sentences. In consequence, the ontologies provided by SEON are a constituting part of our framework. All editing operations for composing queries in HAWKSHAW are defined directly on the SEON ontologies, so that the latter basically define the range of queries that are possible with our approach. However, users do not need to know about the formalism of SEON because they are only exposed to a natural language representation of the ontology, which we added in form of human-readable labels for all classes and properties.

Properties in OWL are a binary relation that can be restricted by specifying domain and range. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, this information can be exploited to filter the verbs that

are allowed to follow a given subject, or the objects that can come after a given verb. In consequence, users are only shown proposals that complete questions in a meaningful way.

The different layers of abstraction provided by SEON enable both specific questions and more abstract ones: When users ask, *e.g.*, *"what persons are contributing to project X,"* they will receive direct instances of the concept *Person* defined in the top-most layer, but also instances of its specializations (*e.g.*, *Developers*, *Committers*, *Testers*, *Reporters* of issues, etc.), which are defined by one of the lower layers. The users can instead exclude non-developers from the result set by entering their question as *"What developers are contributing to project X?"*

In conclusion, we provide a shared taxonomy of important software engineering concepts in Chapter 3. In the next chapter, we explain how to exploit the clear semantics of OWL to translate common developer questions from quasi-natural language to the formal Semantic Web query language SPARQL. This is possible since the natural language annotations of SEON bridge the gap between machine-processable and human-understandable knowledge. The potential impact of SEON is not limited to this thesis. In fact, our ontologies already have been incorporated in a number of other research endeavors. Two of them are mentioned in Chapter 3.

Chapter 4: *Supporting Developers with Natural Language Queries.* In the chapter, we present our framework that allows software engineers to use guided-input natural language strongly resembling plain English to query for information about a software system. The chapter contributes to the thesis a first proof of concept of our approach, as well as an initial case study on its utility.

For the proof of concept, we combine software evolution data provided by EVOLIZER, our platform for software evolution analysis, with Semantic Web technologies for knowledge processing. We focus on queries concerning static source code information, such as *"how often is this field accessed"* or *"what are the subclasses of this class,"* to demonstrate the potential of our approach; but including more data from various software repositories and tools is straight-forward, as we show in Chapter 6. The major components of our proof of concept are listed below:

- **Evolizer Data Layer.** The data layer of EVOLIZER provides a set of of object-relational data models to represent software evolution data, along with adequate importer tools to obtain this data from software repositories. We use a custom-tailored implementation of the FAMIX meta-model [TDD00] to represent facts about source code. A parser then extracts the relevant facts from source code under analysis and populates the model. The parser we use is called ZBINDER and was presented by Pinzger *et al.* in [PGG07]. It is capable of resolving cross references such as method calls and attribute accesses of statements that contain a compile error, which is often the case when the code is incomplete or referenced libraries are missing.

- **Evolizer Ontology Layer.** The EVOLIZER Java implementation of the FAMIX meta-model does not explicitly describe the formal semantics that is needed for automatic knowledge processing tasks such as query answering. To take advantage of Semantic Web technologies, we added an additional layer on top of the data layer by defining an OWL ontology that represents the FAMIX meta-model in terms of OWL classes, relationships and properties. This source code ontology is part of SEON. To populate the knowledge base, we map our Java implementation of the FAMIX meta-model to the OWL ontology. This mapping is done via a custom Java annotation `@rdf`. We manually add an annotation with the URI of the according OWL class to the signature of each Java class that has a counterpart in the ontology. Similarly, we annotate each Java method that should be mapped to a corresponding OWL relation or property name. Java reflection is used to automatically generate RDF statements from Java instances and the resulting triples are stored in a Jena inferencing model.

- **Evolizer Query Layer.** The query layer is based on Ginseng, a **g**uided **i**nput **n**atural language **s**earch **eng**ine, presented by Bernstein *et al.* in [BKKK06]. Ginseng uses a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded. The static part additionally contains information on how to translate query sentences from quasi natural language to SPARQL. To

generate the dynamic part of the grammar, the ontology is loaded into an Apache Jena inferencing model and for each OWL class, individual (instance), object property, and data type property, a grammar rule is generated. The full grammar is then used by Ginseng to guide its users by offering an auto-completion feature, *i.e.*, it presents a popup box with suggestions on how to complete the word that the developer is currently typing into the free-form input field. Once the complete query is concluded by a question mark, it is translated by Ginseng into SPARQL statements and executed against the knowledge base maintained by Jena.

With the case study presented in the chapter, we offer a validation of our approach by addressing the common developer questions that De Alwis and Murphy listed in their paper about the Ferret tool [dAM08]. The two approaches—ours and the one by De Alwis and Murphy—share many similarities in terms of their goals, so that a comparison of their querying capabilities is reasonable. Furthermore, the questions supported by Ferret were identified in two empirical studies [SMDV06, SMV08] to be among the most frequently asked questions by programmers during software evolution tasks and therefore provide a suitable benchmark for our framework. The 36 questions are divided into five categories, namely inter-class, intra-class, inheritance, declarations, and evolution.

Out of the 36 questions, 20 could be answered right away with our approach, seven questions needed to be reformulated or decomposed into multiple other questions. Only nine questions were not yet supported by our tool, mainly because EVOLIZER did not incorporate the necessary data. In particular, we did not store any facts related to the declaration of Eclipse plug-in manifests (five questions) or dynamic program traces (two questions). However, we emphasize that this limitation is not a conceptual one, but has to be attributed to the lack of corresponding fact extractors available for EVOLIZER. Last but not least, queries from the evolution category were supported insufficiently by our proof of concept, because of insufficient data integration. In Chapter 5, we present an algorithm to overcome this insufficiency.

The chapter builds upon the solution to the knowledge representation aspect of **RQ1**, that has been addressed by SEON in Chapter 3, and covers partly the knowledge accessibility aspect. We showed that our approach is feasible and that its querying capabilities are at least comparable to the state of the art. However, in comparison to the latter, our framework allows for greater flexibility in query formulation, as it is only limited by a subset of the English grammar and by the knowledge base that is available. The results of a more comprehensive user study that also incorporates usability aspects is provided in Chapter 6. The knowledge integration aspect of **RQ1** is investigated in the next chapter.

Chapter 5: *Fine-Grained Source Code Change Extraction.* The chapter explains our tree differencing algorithm for fine-grained source code change extraction. Its main contribution to the thesis lies in providing a solution to the knowledge integration aspect of the first research question, **RQ1**. Before we summarize the CHANGEDISTILLING approach, we first explain in the following what exactly we mean by "integration" and why it is important in the bigger context of our thesis.

SEON solves the knowledge representation aspect by offering a formalization of important software evolution concepts and the relations among them (see Chapter 3). We therefore already covered the format we use to describe the knowledge distilled from source code and software repositories such as SCMs or bug and issue trackers. The knowledge integration aspect is concerned with making implicit relations explicit and therefore queryable.

Software repositories are hardly integrated with each other: SCMs contain information about who changed which file and when, but they do not yield any deeper insights into the purpose or semantics of the changes. Common SCMs are further limited to tracking coarse-grained changes based on insertions, deletions, or modifications of single lines—they are unaware of the implicit structure of source code and do not differentiate between individual types of code changes. Research already came up with a number of strategies to overcome the first limitation; integration is achieved by parsing the commit logs for issue numbers in order to link changes to bugs and issues stored in bug databases (*e.g.*, [SZZ05b, FPG03a, FPG03b]). Such heuristics work reasonably well for projects where developers follow

a rigid change process and comment whenever they fix a defect associated with a committed change. Links between issues and file versions raise the possibility to query, for example, for all the bugs that affected a particular file in the past. In many cases, however, the information needs do not end at the file-level. Instead developers are often interested in the fine-grained, syntactical changes applied to classes or even methods, which would normally involve a tedious manual comparison of two versions of a program with the aid of a textual diff tool. The CHANGEDISTILLING approach automates detection of what source code entities were modified in each version of a program. In summary, with our fine-grained change extraction algorithm and a simple bug-linking strategy, we can link code changes to versions of files maintained by the SCM, and then (with the heuristics mentioned above) these versions to bug reports stored in a bug database. This closes the loop, so that domain-spanning queries can be supported. Examples for such queries are:

- What bugs affected this method?
- Which developer changed this class?

CHANGEDISTILLING is an improvement over the existing tree differencing algorithm of Chawathe *et al.* for extracting changes in hierarchically structured data [CRGMW96]. Our improved algorithm is tuned for source code and extracts changes by finding both a match between the nodes of two abstract syntax trees under comparison and a minimum edit script that can transform one tree into the other, given the computed matching.

The main contribution of this thesis to the CHANGEDISTILLING approach is a significant improvement of the matching part of the algorithm. The advances directly affect the computation of the minimum edit script for transforming a version of a program into its succeeding version. In particular, we achieve now a 45% better approximation of the minimum edit script than before. The improved algorithm almost achieves the minimum conforming edit script, which means that we reach a mean absolute percentage error of only 34%, compared to 79% when computing the changes with the original algorithm. Our thesis further makes substantial contributions to the evaluation of the CHANGEDISTILLING approach. For that, we contributed to

a benchmark consisting of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. Last but not least, we provide an ontological representation of the *taxonomy of source code changes* [FG06]. This ontology of fine-grained source code changes is released as part of SEON.

Chapter 6: *Evaluating a Query Interface for Software Evolution Data.* In the final chapter of the thesis, we present the final incarnation of the HAWKSHAW framework, as well as its thorough evaluation. The proof-of-concept implementation described in Chapter 4 was based on our EVOLIZER platform and incorporated an additional export step to translate the object-relational EVOLIZER models into the OWL format that is processed by HAWKSHAW. Recent advances in the Semantic Web, in particular the performance gains of triple stores, allowed us to replace the release history database of EVOLIZER with the file-based Apache Jena TDB triple store. This resulted in a very performant, conceptually clean implementation of HAWKSHAW, in which we directly generate and store RDF triples and skip any other intermediate formats. Furthermore, our framework is no longer limited to queries about source code snapshots, but we have extended our approach substantially by incorporating additional software evolution facets, in particular the development history distilled from SCMs, as well as the bug and issue data obtained from project trackers. Therefore, the chapter ultimately answers the first research question of our thesis, which was concerned with feasibility: yes, it is possible to provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactical complexity.

In Chapter 4, we provided an initial case study on the querying capabilities of HAWKSHAW. The case study left open whether software engineers are able to realize the potential of our approach when they are solving common software evolution and maintenance tasks for a realistic software system. We therefore designed and carried out a user study to investigate this question. With the user study, we pursued two main goals. The first goal was to show that our approach provides an advancement over a set of baseline tools in terms of time efficiency in retrieving the answers, as well as in their correctness. The outcome of this investigation corresponds to the answer of **RQ2**.

The second goal was related to the usability of our approach. In particular, we wanted to answer whether our interface is well-accepted by the users or whether they rather prefer traditional means to access data about their software systems (**RQ3**). For that, we employed standardized satisfaction measures.

We laid out the user study as a *Between Subjects Design* where the subjects were randomly assigned to either an experimental or a control group. The first group was provided with HAWKSHAW. For the second group, we compiled a reasonable set of common developer tools. These tools served as a baseline to compare our approach against. The selection of the baseline is discussed thoroughly in Section 6.4.4 on page 179. The same set of 13 software evolution tasks was then assigned to both groups. We derived the tasks from existing catalogues of common developer questions compiled by other researchers (*e.g.*, in [LVD06, SMV08]). In a nutshell, the results of our user study are:

- Developers achieve at least the same level of correctness when solving common software evolution tasks with our quasi-natural language approach as with the baseline of existing tools, *i.e.*, a common Java IDE and the Web front-end of a widely used bug and issue tracker.
- Our approach leads to a significant improvement in time efficiency. Overall, we observed time savings of 35.41% when compared to the baseline, with gains of up to 300% in some particular cases.
- The overall system satisfaction of users with HAWKSHAW is clearly better than the one of those that used the baseline tools. The subjects rated our approach on average with a total score of 74.31 on the System Usability Scale [Bro96], whereas the baseline only achieved an average rating of 38.24. The high score of HAWKSHAW is directly related to its high usability and learnability.

Based on the empirical results, we can answer **RQ2** and **RQ3** positively: Users are indeed able to successfully answer common developer questions with our quasi-natural language interface. Overall, HAWKSHAW provides a clear advancement over the baseline tools in respect to both correctness and time efficiency. It is further well-accepted, and the significantly higher system

satisfaction score emphasizes that our novel approach is also preferred over traditional tools.

1.5 Summary of Contributions

As software systems get more complex, efficient tools to support software engineers during their development and maintenance tasks become more important. Modern IDEs already made a great leap forward in providing a variety of features to solve various software development and maintenance tasks. Unfortunately, powerful features often come with an increase in complexity. This can put a significant cognitive burden on developers and, in some cases, it may even be easier to solve a task manually than to master a tool. Although experienced developers usually know exactly what information they are looking for, they often do not know how to get to it. They simply do not know how to turn conceptual queries into something their IDE understands.

In consequence, developers often spend as much as 60% to 90% of their time reading and navigating code and other data sources such as software repositories [Erl00]. Although these problems are known, the user interface paradigm of IDEs underwent no fundamental changes for literally decades and only few researchers such as Bragdon *et al.* with their Code Bubbles metaphor [BRZ+10] have begun to rethink it in order to reduce the effort substantially.

In this thesis, in contrast to the Code Bubble approach, we did not turn away fundamentally from the successful and well-proven paradigm that prevails in modern IDEs such as Eclipse or Microsoft Visual Studio. Instead, we strived for leveraging and complementing the existing tools carefully with HAWKSHAW, a well-integrated, single point of access for common information needs of developers and maintainers alike. By incorporating a guided-input natural language query interface and Semantic Web technology, we provided a valuable alternative to menu-driven search interfaces of common IDEs and project tracking tools. We argued that our HAWKSHAW approach is helpful in solving various tasks related to software evolution and maintenance, and that it also scales to real industrial-size software systems. As a proof of concept, a fully functional prototype of HAWKSHAW has been implemented and evaluated.

We investigated the question how software evolution knowledge can be ade-

quately represented by means of ontologies. As an answer to this question, we presented SEON, a family of ontologies that model many different facets of a software's life-cycle to provide a shared taxonomy of important software engineering concepts. We have made them available to other researchers and they already have found multiple applications besides the one presented in this thesis (*e.g.*, in [GG11, MFGW12]).

By exploiting the clear semantics of OWL we established a queryable knowledge base of software evolution facts based on SEON and integrated it into the Eclipse IDE. We explained how various questions formulated by developers in quasi-natural language can be automatically translated to the formal Semantic Web query language SPARQL. This was possible, since the natural language annotation layer of SEON bridged the gap between machine-processable and human-understandable knowledge.

A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. Previous approaches that investigate source code changes relied on information provided by versioning systems such as CVS or SVN. They track changes of source code files on a *text basis* without storing detailed information. The CHANGEDISTILLING approach automates detection of the source code entities that were modified in each version of a program. Based on the contributions made in the context of this thesis, the change detection accuracy of CHANGEDISTILLING could be increased by 45%. The increase results from a significant improvement of the matching part of the algorithm. With our fine-grained change extraction algorithm and an existing bug-linking strategy, we were then able to link code changes to versions of files maintained by a SCM, and then these versions to bug reports stored in a bug database. This provided tight integration between data that was previously stored in independent, hardly queryable information silos. Our achievement therefore de-facto enabled the execution of cross-repository queries in quasi-natural language.

With statistical significance, we have shown in a user study with 35 subjects that all these components together constitute a framework that allows software engineers to deal more efficiently with common software evolution and maintenance tasks. In particular, we observed that developers achieve at least the same level of correctness when solving the tasks with our quasi-natural language approach as with a baseline of existing tools. Furthermore, in comparison with them, our approach leads to a significant improvement in time efficiency. Overall,

we observed time savings of 35.41% over the baseline, with gains of up to 300% in some particular cases.

Not only did developers succeed in solving various important tasks with our framework—their overall system satisfaction was also clearly better than that of the baseline's users. The subjects rated our approach on average with a total score of 76.11 on the System Usability Scale, whereas the baseline only achieved an average rating of 33.09. The high score of HAWKSHAW is directly related to its high usability and learnability.

The user study focussed on less experienced subjects since we expected our approach to be especially apt for novice developers, not yet deeply familiar with the tools available. The positive feedback received from some of the more seasoned participants provides anecdotal evidence that the results of the study can be generalized to a broader target group of users. Further investigations are needed to support this claim with scientific evidence.

The HAWKSHAW framework revolves around a knowledge base built with Semantic Web technologies and a quasi-natural language interface. The Semantic Web yet struggles to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (*e.g.*, in [KSG+10]). The same accounts for natural language interfaces, which have recently gained momentum in other domains. Popular examples are Apple's Speech Interpretation and Recognition Interface, (Siri)[9] the Wolfram Alpha answer engine developed by Wolfram Research,[10] and IBM's Watson/DeepQA computer system for answering natural language questions [FBCC+10, Fer11]. However, also natural language interfaces have been mostly neglected in software evolution research so far.

The conclusion we can draw from the strong empirical results found during the user study presented in this thesis is that both the Semantic Web and natural-language interfaces exhibit significant potential for building the next generation of software engineering support tools. They should therefore be at least considered whenever researchers in the field of software engineering devise approaches that involve knowledge representation and developer-computer interfaces. A quasi-natural language interface such as the one incorporated in our HAWKSHAW

[9] http://www.apple.com/iphone/features/siri.html
[10] http://www.wolframalpha.com/

framework requires basically no learning effort from its users and therefore can accelerate the adoption of novel research tools in practice.

1.6 Limitations and Future Work

We demonstrated that our HAWKSHAW approach is both flexible and extensible, and that it also provides developers a statistically significant benefit over existing tools. However, we still identified several areas for improvement and opportunities for future research. In the the remainder of this section, we therefore elaborate on the limitations of our approach, as well as on the work necessary to overcome them in the future.

- **Grammar Rules and Synonyms.** Both the case study in Chapter 4 and the user study presented in Chapter 6 demonstrated that a surprisingly small set of static grammar rules and synonyms allows for a variety of different queries. However, additional investigations are needed to identify variations in the exact phrasing of conceptual queries that might occur when software engineers formulate their information needs in practice. The findings then need to be encoded in terms of static grammar rules. Furthermore, the natural language annotations (synonyms) of SEON are based on our personal vocabulary. This vocabulary might be biased towards the programming languages and tools we regularly use and therefore fail to adequately describe the concepts which developers with a different background are familiar with.

  We evaluated the use of general-purpose lexical databases of English, in particular the WordNet database [Mil95], to overcome this limitation. However, for our approach, they have proven themselves unsuitable. The problem we encountered was that many technical terms have also non-technical meanings in daily life. For example, the term "Method" used in object-orientation has synonyms such as "adjustment," "approach," "fashion," etc. If we automatically add those to the list of proposals presented by our query interface, then the developers are no longer restricted to reasonable questions, *i.e.*, they can then enter completely meaningless ones such as *"What fashion invokes the approach foo()?"* A database of technical software engineering terms might alleviate the issue.

- **Additional Information Domains.** We see potential in integrating dynamic traces, additional metrics, and test or build-specific data into our knowledge base. But also the addition of application-domain-specific knowledge might be worthwhile, along with a representation for *concerns*, *i.e.*, everything considered as a conceptual unit, such as features, nonfunctional requirements, and design idioms [RM07]. This would move our approach closer to LaSSIE presented by Devanbu *et al.* in [DBS91]. We presented an extensible platform for RDF-based read and write access to relational databases in [HGWG11]. The platform facilitates the incorporation of data from existing approaches into our knowledge base since it can act as mediator between the HAWKSHAW query framework and existing relational databases of software evolution data.

- **Temporal Queries.** Our approach is currently not particularly well-suited for answering questions related to time intervals or specific points in time. For example, questions such as *"What classes were changed yesterday?"* or *"What bugs were fixed between May and August?"* cannot be entered directly. However, it is possible to query, *e.g.*, for all bug fixes and then sort the results by their fix date. This puts HAWKSHAW on a par with many tools used in practice, but formal query languages would clearly have an edge over our approach in that respect (at the cost of additional learning effort).

  While it is notable that existing catalogues of common developer questions rarely contain examples such as the ones mentioned above, we still know from our own experience that they occur frequently in practice, so that an adequate support would be desirable. While the static grammar of HAWKSHAW can be extended to incorporate corresponding natural language rules, more research is needed to come up with an appropriate translation into SPARQL. Approaches such as Temporal Reasoning could provide a solution to this issue [Tap11].

- **Textual Search**. Currently it is not possible to perform a textual or keyword-based search because of the guided nature of the query composition approach. In consequence, users cannot search, for example, for all the files that contain the word "database." This is sufficiently well supported by existing tools so that we, in principle, see no immediate need for action. However, to preserve the idea of a single point of access for common information

needs, it would still make sense to add special non-terminal symbols to the static grammar rules that would temporarily switch the query interface from a guided mode into one that allows for entering free-form text (*e.g.*, when entering opening quotation marks until closing ones are typed). The translation into SPARQL is then straight-forward thanks to built-in regular expression support.

- **Search Result Presentation**. Results to queries performed with HAWKSHAW are presented in a rather conservative way, *i.e.*, in tabular form. However, we can envision a more visual approach based on visualization techniques to highlight the results in their context. We already started to explore the possibilities with our ONTOX approach presented in [Zeh11]. Proper integration followed by an evaluation are the next steps.

- **Study Setting**. A last limitation is concerned with the artificial study setting used for evaluating the HAWKSHAW framework. While we followed carefully the scientific method to prove the value of our approach, a lab study cannot be a replacement for a thorough field study with real developers working on real projects in industry. From such a study, we expect to gain deeper insights on the potential and practical limitations of our query interface and how it can be embedded in a regular software engineering process. Future research should also investigate to what extent the level of development experience influences the acceptance of our approach by its potential users.

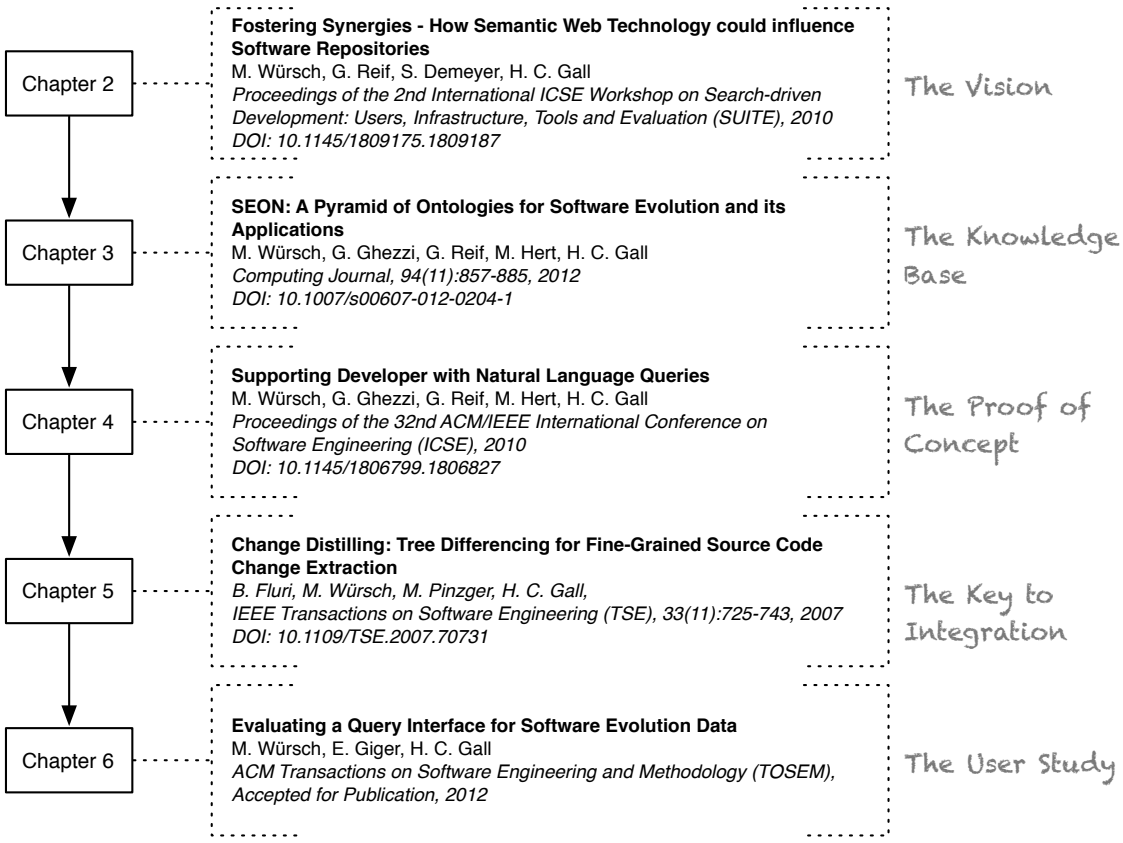


Figure 1.4: Thesis Roadmap

1.7 Roadmap of the Thesis

The remainder of this thesis consists of five more chapters: Chapters 2 to 6. Each chapter is based on a scientific publication, as depicted by the overview given in Figure 1.4. Together, these publications form the contribution of our thesis.

Chapter 2 presents our long-term vision for the next generation of software repositories. We particularly argue that ontologies are a valuable means for representing knowledge from the domain of software evolution and maintenance. The chapter therefore motivates our decision to build HAWKSHAW on the backbone of the Semantic Web.

In Chapter 3, we present SEON, our family of software evolution ontologies. These ontologies represent the schema of the knowledge base which our quasi-natural language query interface operates on.

An initial proof of concept for our approach is given in Chapter 4. We present a first prototype which does not exploit the full potential of the final HAWKSHAW approach, but already guides developers in composing and executing quasi-natural language queries about static source code information. In a first case study, we compare the querying capabilities of another state-of-the-art research tool to those of our approach. The comparison demonstrates that our framework can be used to answer some of the most common program comprehension questions that arise during software evolution tasks.

Chapter 5 describes the CHANGEDISTILLING approach for identifying fine-grained change types between program versions, along with an elaborate evaluation of the tree differencing algorithm used to extract and classify the changes. The fine-grained change information is later used in the final incarnation of our HAWKSHAW framework to complete the facts stored in our SEON knowledge base. The additional knowledge allows users of our approach to perform a broader range of more specific queries since it bridges the gap between line-based textual change information tracked by SCMs and the actual syntactical changes applied to source code by its developers.

Our thesis culminates in Chapter 6, where we present the final HAWKSHAW framework and its evaluation. In contrast to the proof of concept provided in Chapter 4, we incorporate not only the static source code information, but also integrate it with other knowledge distilled from SCMs or bug and issue trackers. For the evaluation part, we elaborate on how we designed and conducted a user study with the aim to compare HAWKSHAW against a set of baseline tools. A sound empirical analysis of the study results concludes the chapter.

Fostering Synergies

How Semantic Web Technology could influence Software Repositories

2

Fostering Synergies: How Semantic Web Technology could influence Software Repositories. M. Würsch, G. Reif, S. Demeyer, H. C. Gall, Proceedings of the 2nd International ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE), 2010
DOI: 10.1145/1809175.1809187

The state-of-the-art in mining software repositories stores software artifacts from various sources into monolithic relational databases. This puts a lot of querying power in the hands of the software miners, however it comes at the cost of enclosing the data and hamper cross-application reuse. In this paper we discuss four problem scenarios to illustrate that Semantic Web technology is able to overcome these limitations. However, it requires that the software engineering research community agrees on two prerequisites: (a) a common vocabulary to talk about software repositories—an ontology; (b) a strategy for generating unique and stable references to all software artifacts inside such a repository—a Universal Resource Identifier (URI).

2.1 Introduction

Over the last decade, the software engineering community developed various tools which help engineers to specify, develop, test, analyze, and maintain software. Most of these tools use proprietary data formats to store their artifacts. This hampers tool-interoperability and renders querying difficult, especially when you want to query across tool domains. Queries such as "In which release was this bug fixed and which source code modifications where necessary to fix it?", however, involve several domains (*i.e.*, static source code, version control, issue tracking).

The *Mining Software Repositories* community has tackled this issue by mirroring software artifacts from various sources in a central (relational) database [DGLP08]. This additional querying power gave rise to numerous experiments where researchers successfully mined such databases for interesting patterns (see [KCM07] for an overview; specific examples can be found in [FPG03b, BEJWKG05, SZZ05b, GFP09]). Unfortunately, such a central database imposes a universal data schema onto all contributing tools, turning the software repository into a rigid and inflexible monolith. Especially when integrating tools supplied by different research groups, such a software repository is nothing more than the kind of stovepipe systems we all resent.

Semantic Web technology has been designed as a solution to such integration problems. In a nutshell, it provides a standardized, well-established framework that allows data to be shared and reused across application, enterprise, and community boundaries. It does so by means of two concepts: (a) ontologies and (b) Universal Resource Identifiers (URIs). The former provides the formal vocabulary that applications can use to exchange semantically rich data, by defining the entities in the domain of discourse and the relationship between them. The latter is a unique and stable reference to all possible entities which enable hyperlinks between semantically-annotated data in one place with data in other places.

In this paper we argue that the use of Semantic Web technology enables the construction of highly interlinked and distributed knowledge bases, which form the basis for flexible queries and data analysis. Using four scenarios, we demonstrate several problems with the current state of the art, *i.e.*, centralized databases. For each of the scenarios we also show how Semantic Web technology may come to the rescue. We conclude the paper with a research agenda which lists the prerequisites that need to be resolved before these scenarios can be realized.

2.2 The Semantic Web in a Nutshell

The Semantic Web was designed to be an extension of the Web as we know it today, enriching it with meta data describing the semantics of Web pages to make their content computer-processable. To describe information on Web pages with meta data accordingly, an *ontology* has to be defined that formally describes the concepts (classes) found in the domain of discourse, the relations between these concepts and the properties used to describe them [Gru93]. These principles are not restricted to Web pages but can be applied to any kind of data. In the software engineering domain, for example, we can define concepts, such as *User*, *Developer*, *Bug*, *Module*; relationships, such as *reports bug*, *fixes bug*, and *is assigned to bug*. Since the Semantic Web describes this information based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place.

The Resource Description Framework (RDF) [Ke04] is the data-model for representing meta data in the Semantic Web. The RDF data-model formalizes meta data based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource in the universe of discourse. A resource can be almost anything: a bug report, a person, a Web page, a CD, a track on a CD, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. The data-model of RDF is a graph where the subject and object are the nodes and the predicate is a labeled, directed arcs pointing from the subject to the object. The query language SPARQL [Pe08] can be used to query such RDF graphs.

2.3 Scenarios

In the following, we list typical problem scenarios in the context of software analysis and software repository infrastructure. For each problem, we identify the key challenges it brings and elaborate on traditional solutions and their shortcomings, before we outline how the Semantic Web could resolve the issue. In Section 2.4, we set up a research agenda our community needs to work on.

Scenario 1: A Shared Vocabulary

Description: *Alice has just started a Ph.D. having the working title "On the Influence of the Programming Language on the Occurrence of Bugs in Open Source." In particular, she is curious about whether certain language features are misleading developers to introduce defects. After a thorough discussion with her promoter, she realizes that she needs to build up a query-able knowledge base containing a substantial amount structural information about source code of various open source project, as well as related bug reports. She plans to investigate projects written in Java and C# first, as the languages share many similarities. She reasons about a suitable meta model to represent her data and whether there are already existing parsers that she could adapt, so that she would not need to start from scratch.*

Key Challenges: One of the critical design aspects when building a knowledge base is to define a meta model that describes the data in an adequate level of detail. To share data among different tools and knowledge bases, they need to understand the same vocabulary.

Traditional Approaches: In practice, there are a number of meta models, for example for source code, that define the same concepts, but name them differently. For example, *C++ Data Model* [CGK98] of Chen and the *FAMOOS Information Exchange Model (FAMIX)* of Tichelaar *et al.* [TDD00] can both be used to describe source code written in C++. Although they share many commonalities, tools written to work on FAMIX can not process instances of Chen's model and vice versa. Further, the meta models are often implemented in terms of a relational database schemas. Exchanging schemas among different databases, however, is relatively uncommon, due to vendor-specific implementations of data definition languages. Instead, and despite the advent of specialized exchange formats, such as XMI [Obj98] or GXL [WKR02], data is often serialized into plain XML or a comma separated value (csv) format. These formats are not semantics-preserving

and therefore of limited use.

The Semantic Web Approach: The Semantic Web provides a framework to structure data so that it becomes machine processable. The Web ontology language OWL is used to model the vocabulary of a domain [De04] by describing properties and classes; for example, relations between classes (*e.g.*, inheritance relationships, disjointness), cardinalities, equality, or characteristics of properties (*e.g.*, inverse, symmetry). These vocabularies are called ontologies and have explicit formal semantics. While it is uncommon to exchange relational database schemas, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged without the loss of data semantics.

Scenario 2: Linked Data

Description: *When Alice presents her first workshop paper, she rans into Bruce who is a Ph.D. student too. Bruce is focussing on bug prediction and has also put online a knowledge base, already containing hundreds of systems written in Smalltalk. Alice realizes that the work of Bruce is likely to provide complementary insights on her own research and convinces her promotor to set-up a one week research visit to investigate whether it would be possible to merge the two databases.*

Key Challenges: To interlink repositories on the Web, a flexible data model is needed that allows to expose references to data stored within a repository to the outside. It further should provide means to describe relationships between entities stored in different locations. These references need to remain stable over time to guarantee data consistency across the repositories.

Traditional Approaches: With classical relational database technology alone, synergies between research tools are hard to exploit. For example, we cannot simply establish connections between data stored in EVOLIZER [GFP09] and Sourcerer [BOL09], as it is not possible to set a link from one repository to another—relations are local, not universal. Cross-domain queries spanning multiple repositories are impossible.

The Semantic Web Approach: While the Semantic Web is no panacea, one of its driving forces is the basic assumption that data becomes more useful the more it is interlinked with other data. The simple but powerful concept of statements represented by triples of URIs can be used to build an internet-scale graph of information because it enables us to link and query data that is stored in different

locations. Queries on this graph are also possible, thanks to the query language SPARQL.

Let us assume that we build our repositories the way that they generate and expose an URI for every software artifact that they store, and that these URIs are also de-referenceable over the Internet. Then consider a Java class called `Foo` to be stored in one repository—maybe Sourcerer, a search engine for open source code—together with other classes and data about the relationships between them. Assume further that a particular bug report with the number 124 is hosted in a completely independent repository, for example, a Bugzilla or Jira issue tracker, along with other reported bugs, information on their interdependencies, severity and priority, attachments and related discussion threads from bug reporters and developers, and so on. A statement stating that the bug #124 affects the `Foo` class can then be stored in a third repository, possibly our release history database called EVOLIZER. The following s-p-o triple shows how such a statement could look like:

```
http://bugzilla.myProject.org/bugs/nr124
http://myBugOntology.org/affects
http://sourcerer.ics.uci.edu/myProject/Foo.java .
```

Given this triple, client applications, as well as humans, can easily follow the links to access the raw resources and use either the subject, object, or even the predicate as an input for further SPARQL queries.

Scenario 3: Traceability of Results

Description: *Alice is working together with Bruce on her first journal publication describing an empirical study that they have conducted together. To allow other researchers to replicate their results they describe precisely which system releases they used for their study, using the sentence "To demonstrate that our novel approach shows good prediction performance, we selected the five most recent releases from ArgoUML . . . ". Moreover, Alice stores all the data, as well as the detailed results, into a comma separated file, places it on a Web server. In her paper, she lists a link referring to the replication package.*

Key Challenges: The origin of scientific results should be traceable and independently replicable. It has to be made clear what data has been analyzed, *i.e.*, meta data needs to be supplied along with the results. For example, which Java classes were analyzed in particular? Which versions were considered? Was

only code on the trunk of a version control system considered, or code in all branches equally? The analysis itself needs to be described in similar detail. What preprocessing steps were applied? Was there any filtering necessary? How did the queries look like exactly?

Traditional Approaches: Publications on empirical studies or machine learning applications in the context of software evolution usually list releases of systems being investigated. In addition, authors often provide input and output files for statical tools, such as R.[1] On the one hand, these files do not contain any raw, but rather preprocessed—and therefore potentially biased—data. This makes it harder to extend studies of other researchers, because, while preprocessing, subtle differences due to implementation issues can accumulate. On the other hand, the provided input and output files are in the xml or csv format and, as such, do not contain any semantics. This potentially makes them harder to reuse in other tools.

The Semantic Web Approach: In mature scientific fields, authors of publications are requested to provide enough information to allow their findings to be replicated by someone else working independently. Same counts for software engineering research and the Semantic Web may contribute in that respect. RDF statements and URIs describe data under analysis more accurately than sentences like *"To demonstrate that our novel approach shows good prediction performance, we selected the five most recent releases from ArgoUML . . . "* In addition, authors can make the SPARQL queries public that they have used in their analysis and, since the results are represented again as RDF statements, they can directly be processed in other tools for further investigations, *e.g.*, as input for another query.

Scenario 4: Relations

Description: *Alice has finished her Ph.D. and managed to build an integrated repository to store and query vast information about source code, related bugs, and—thanks to a couple of busy master's students—also requirements documents, such as use case descriptions, and even data about developers,* e.g., *their code ownership and social network connections. Bruce is more than impressed and starts to explore the repository by querying. In particular, he is interested in finding all kinds of relations between developers and particular Java classes.*

[1] `http://www.r-project.org/`

```
SELECT
      ?relation
WHERE {
      developer:id101 ?relation javaclass:DBaccess . }
```

Listing 2.1: SPARQL Query to retrieve the Relations between a Developer and Java Class

Key Challenges: The quality of a meta model has direct impact on the effectiveness of search interfaces: the meta model (as well as the query language used to query the model) needs to be expressive to allow a broad range of queries and its structure needs to be simple to keep the complexity of common queries, as well as the cost of executing them, low.

Traditional Approaches: There is no consistent way to get the meaning of a relation in relational databases. In fact, a query can join tables by any columns which match by data type – without any check on the semantics. While humans can often guess the meaning of a relation, computers can not. They need to be supplied with additional information. It is therefore necessary to encode a significant amount of implicit knowledge into our applications to make use of the data. To search in an existing repository or to build an own tool on top of it, researchers need to be aware of, and understand this implicit semantics.

The Semantic Web Approach: The SPARQL query language allows, in particular, to query explicitly for relations among resources. Consider the example query given in Listing 2.1 that selects all direct relations between a given developer and a particular Java class.

The basic graph pattern in the example query consists of a single triple pattern with one variable (?relation). The triple pattern matches all triples where the subject is the developer with the id 101 and the object is a Java class called `DBaccess`, respectively. Each solution gives one way to bind an RDF property to the ?relation variable. The SELECT clause specifies that the ?relation variable is returned as query result. This example demonstrates a SPARQL query which returns the relation between two resources.

These kind of queries are impossible in the relational and in the object-oriented paradigm, unless relationships are explicitly mapped to tables or, in the case of object-orientation, modeled as association classes. The latter, however, can make them difficult to distinguish from "real" classes. Given the high importance of

relationships in software engineering, it would be preferable to model them as first class objects—which is exactly what the Semantic Web does.

2.4 Research Agenda

To take advantage of the Semantic Web like envisioned in the scenarios above, the software engineering community should address the following challenges.

Challenge 1: Formalize a common vocabulary to describe software artifacts and their relationships in terms of an ontology. It is relatively uncommon to exchange relational database schemas among different databases but ontologies were explicitly designed to be shared. This also means that one should re-use existing vocabularies whenever possible to enable client applications to process and search in data from many different sources more easily. It is also common practice to mix terms from existing vocabularies. Several ontologies for the domain of software engineering already exist. Some examples are: *Description of a Project (DOAP)*, [2] a vocabulary to describe software projects, and in particular open source. The *Bug And Enhancement Tracking LanguagE (baetle)* [3] describes information kept in bug databases and re-uses, among others, the DOAP ontology. EvoOnt [KBT07] and our *Software Engineering Ontology (SEON)*[4] both define a vocabulary to represent information found in version control and issue tracking systems. They also include an ontology for Java source code. This non-exhaustive list of existing ontologies can serve as good a starting point but also needs consolidation and refinement, driven by real scenarios and applications developed by researchers in our domain. In other words, it is a community effort to define an ontology, since it manifests the common, shared data model to represent the data a domain.

Challenge 2: Devise strategies to generate URIs for artifacts of software projects. While redundancy is generally less a problem in the Semantic Web, compared to the relational paradigm, it is crucial that every software artifact can be uniquely identified—and that these URIs can be dereferenced and remain stable over time. Further, a strategy for generating an URI needs to be deterministic and reproducible by other tools. This means that if we parse, for example, the contents

[2] `http://trac.usefulinc.com/doap`
[3] `http://code.google.com/p/baetle/`
[4] `http://evolizer.org/`

of a version control system twice and, in each pass, generate URIs for the source code artifacts found, then the URIs of each pass need to be identical.

2.5 Conclusions

In this paper, we have explained how Semantic Web Technology can be used to construct the next generation of software repositories. In our vision, software repositories should blend into a query-able global information space of interlinked data about all possible software engineering artifacts. However, it requires the software engineering community to agree on two issues: (a) an ontology to describe the software artifacts and their relationships; (b) a strategy for generating URIs for such software artifacts.

In the meantime, what should happen to the existing repositories? Indeed, research groups have invested a significant amount of effort in building these software repositories: it is unrealistic—but, fortunately, completely unnecessary—to throw them away. A tutorial on *How to Publish Linked Data on the Web* can be found in [BCH10]. The tutorial also features a section on how to publish existing relational database as *Linked Data* and lists several tools to do so. So migration strategies that allow for an incremental adoption of ontologies and URIs are entirely feasible, and we encourage the research community to make the necessary preparations.

SEON

3

A Pyramid of Ontologies for Software Evolution and its Applications

SEON: A Pyramid of Ontologies for Software Evolution and its Applications. M. Würsch, G. Ghezzi, M. Hert, G. Reif, H. C. Gall, Computing Journal, 94(11):857–885, 2012
DOI: 10.1007/s00607-012-0204-1

The Semantic Web provides a standardized, well-established framework to define and work with ontologies. It is especially apt for machine processing. However, researchers in the field of software evolution have not really taken advantage of that so far. In this paper, we address the potential of representing software evolution knowledge with ontologies and Semantic Web technology, such as Linked Data and automated reasoning. We present SEON, a pyramid of ontologies for software evolution, which describes stakeholders, their activities, artifacts they create, and the relations among all of them. We show the use of evolution-specific ontologies for establishing a shared taxonomy of software analysis services, for defining extensible meta-models, for explicitly describing relationships among artifacts, and for linking data such as code structures, issues (change requests), bugs, and basically any changes made to a system over time. For validation, we discuss three different approaches, which are backed by SEON and enable semantically enriched software evolution analysis. These techniques have been fully implemented as tools and cover software analysis with web services, a natural language query interface for developers, and large-scale software visualization.

3.1 Introduction

Scientia potentia est. Knowledge is power. For millennia this maxim has been valid, and will likely remain so in the future—even in an age of information overload, where the entire humankind produces roughly two zettabytes data a year.[1]

This also holds for the domain of software engineering, where even small development teams accumulate gigabytes of interdependent artifacts over the years. They are stored in software repositories, such as version control systems, issue trackers, but also in Wikis, and even mailing lists. Understanding what factors distinguish successful development projects from others is key to improve the quality of software systems. Distilling the knowledge of best practices from random noise found in a software repository is what the field of software evolution research and mining software repositories aims for.

But data is not necessarily information, and information not necessarily knowledge. Successful differentiation requires understanding of data semantics and interpretation. The obvious solution to this dichotomy is that machines and humans form a joint-venture: humans define the semantics and machines bring in their computational power for the advent of the next generation of software evolution support tools. The Semantic Web provides the instruments to achieve such a synergy; ontologies created by human beings represent knowledge and give semantic meaning to raw data so that machines can automatically process and exchange it. Reasoners make implicit knowledge explicit by inferring relations that were previously missing. Interestingly, these technologies yet struggle to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (*e.g.*, in [KSG+10]).

In this paper, we pursue the research question, how we can adequately describe software evolution knowledge by means of ontologies. This includes knowledge about stakeholders, activities, artifacts, and the relations among all of them. The ultimate goal is to provide software engineers with effective tool-support for managing software systems over their entire life-cycle.

The contributions of our paper are threefold:

1. We critically reflect on the potential that the Semantic Web yields for software

[1] According to the study "Digital Universe: Extracting Value from Chaos" by IDC, humans created 1.8 zettabytes data in 2011. This value is estimated to double every two years.

evolution. In particular, we show four characteristics that are most beneficial for the field: shared taxonomies, extensible meta-models, explicit relations, and Linked Data.

2. We present SEON, our family of software evolution ontologies. These ontologies describe knowledge on multiple levels of abstraction ranging from code structures up to stakeholder activities.

3. We describe three semantics-aware tools that make extensive use of SEON and help developers in dealing with large amounts of software evolution data: software analysis with web services, a natural language query interface for developers, and large-scale software visualization. All three of them have been fully implemented for a proof of concept.

In the remainder of this paper, we will describe the potential of Semantic Web technology for dealing with software evolution.

In Section 3.2, we give a brief overview on the Semantic Web and related technologies, before we discuss in Section 3.3 the advances they can bring to the field of software evolution research. We also address a set of general challenges yet to be solved before the full potential of Semantic Web-enabled approaches can be realized.

At the core of this paper is SEON, our pyramid of ontologies for software evolution, which is described in Section 3.4. These ontologies provide a taxonomy to share software evolution data of various abstraction levels across the boundaries of different tools and organizations.

In Section 3.5, we describe three different applications of SEON from three distinct domains to showcase the utility and versatility of ontologies in the context of software evolution research. A selection of other ontology-driven approaches in the field of software engineering is discussed in Section 3.6. In Section 3.7, we conclude the paper.

3.2 The Semantic Web in a Nutshell

Berners-Lee *et al.* define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* [BLHL01]

Despite its origins, the Semantic Web is not limited to annotating webpages with meta-data. Virtually any piece of knowledge can be described in a computer-processable way by defining an ontology for the domain of discourse. An ontology formally describes the concepts (classes) found in a particular domain, as well as the relationships between these concepts, and the attributes used to describe them [Gru93]. For example, in the domain of software evolution, we define concepts, such as *User*, *Developer*, *Bug*, or *Java Class*; relationships, such as *reports bug*, *resolves bug*, or *affects Java Class*; and attributes, such as *email address of developer*, *resolution date of bug*, *severity of bug*, etc.

Since the Semantic Web describes knowledge based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place. The data representation format no longer needs to be custom-tailored to a specific task, but can be re-used later.

Researchers and practitioners came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases to pursue the vision of the Semantic Web. The Resource Description Framework (RDF) [Ke04] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a project, a bug report, a person, a Web page, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model, information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [Pe08] can be used to query such RDF graphs.

RDF itself is domain-independent in that no assumptions about a particular do-

main of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [De04]. OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [PSHe04]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class. OWL can describe both uniformly, data schema and instance data.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. Jena[2] emerged from the *HP Labs Semantic Web Program* and recently became an Apache incubator project. It is a Java framework for building applications for the Semantic Web and provides a programmatic environment for RDF and OWL. Reasoners, *e.g.*, Pellet [SPG+07] or HermiT [SMH08], infer logical consequences from a set of asserted facts or axioms. RDF databases, such as Sesame[3] or Virtuoso,[4] store RDF triples and can be queried with SPARQL.

3.3 The Potential of Ontologies in Software Evolution Research

Over the last decade, software evolution research brought up various tools that help engineers to better deal with large, ever-changing legacy systems. We argued in [WRDG10] that most of these tools use proprietary data formats to store their artifacts, which hampers tool-interoperability. Furthermore, querying software evolution knowledge is difficult, especially when queries span across different domains. Queries such as *"In which release was this bug fixed and which source code modifications where done to fix it?"* involve several domains (*i.e.*, static source code, version control, issue tracking), something which is not originally supported by common software repositories.

The *Mining Software Repositories*[5] community tackled this issue by mirroring software artifacts from various sources in a central (relational) database [DGLP08].

[2] `http://incubator.apache.org/jena/`
[3] `http://www.openrdf.org/`
[4] `http://virtuoso.openlinksw.com/`
[5] `http://www.msrconf.org/`

This gave rise to numerous experiments where researchers successfully mined such databases for interesting patterns (see [KCM07] for an overview; specific examples can be found in [BEJWKG05, FPG03b, GFP09, SZZ05b]). Unfortunately, such a central database imposes a universal data schema onto all contributing tools, turning the software repository into a rigid and inflexible monolith.

Semantic Web technology has been designed as a solution to such integration problems. In the following, we briefly revisit the characteristics of the Semantic Web that we identified in in our previous work to be most beneficial for the field of software evolution research.

3.3.1 Establishing a Shared Taxonomy of Software Evolution

One of the critical design aspects when building a knowledge base is to define a meta-model that describes the knowledge in an adequate level of detail. To share data among different tools, they need to understand the same vocabulary.

In practice, there are a number of general-purpose meta-models in software engineering, such as the Dagstuhl Middle Metamodel (DMM) [LTP04], as well as more specific ones, *e.g.*, for source code. Many of them define the same concepts, but name them differently. The *C++ Data Model* [CGK98] of Chen and the *FAMOOS Information Exchange Model (FAMIX)* of Tichelaar *et al.* [TDD00] can both be used to describe source code written in C++. Although they share many commonalities, tools written to work on FAMIX cannot process instances of Chen's model and vice versa, *e.g.*, to replicate experiments. Further, meta-models are often implemented in terms of a relational database schema. Exchanging schemata among different databases, however, is relatively inconvenient, due to vendor-specific implementations of data definition languages. Instead, and despite the advent of specialized exchange formats, such as RSF [MK88], XMI [Obj98], or GXL [WKR02], data is often serialized into plain XML or a comma separated value (csv) format. These formats are not semantics-preserving and therefore of limited use.

While relational database schemata are hardly ever exchanged, ontologies were explicitly designed to be shared. They can be serialized using the RDF/XML standard and exchanged without loss of data semantics. In Section 3.4, we propose our set of ontologies that provide a taxonomy for important concepts in the

domain of software evolution. With the approach described in Section 3.5.1, we demonstrate how such taxonomy fosters interoperability between an entire ecosystem of software services.

3.3.2 Defining Extensible Meta-Models

Especially in a research context, meta-models tend to evolve constantly. Therefore, they need to be designed to be extensible. For example, adding data about additional software artifacts should be straight-forward and possible without breaking applications that rely on the original model.

When meta-models are extended, this usually enforces database schema changes. This is a time consuming operation, as the whole repository and all database keys have to be reorganized. Chances are more than likely that existing applications directly accessing the database will break in such a case.

Designing ontologies is comparable to designing Entity-Relationship or UML models. The result is a data schema. In the Semantic Web, however, the schema itself is described in terms of RDF triples, making it more flexible to changes than the relational one. No distinction between data and ontology is necessary, as both are simply additions or deletions of triples. It is therefore unproblematic to add more ontologies and to specialize existing concepts and properties by deriving sub-concepts and sub-properties.

In Section 3.5.2 we present a query approach that especially benefits from the extensibility of ontologies, as well as from the fact that data and meta-data are represented uniformly. Our query system analyses both, the data and meta-data and uses the results to guide developers in composing and executing queries related to program comprehension tasks. When we add new ontologies to SEON, our query system is able to deal with this additional knowledge without requiring us to change a single line of code.

3.3.3 Making Relations Explicit

There is no consistent way to get the meaning of a relation in relational databases. In fact, a query can join tables by any columns, which match by datatype—without any check on the semantics. While humans can often guess the meaning of a relation, computers can not. They need to be supplied with additional information.

It is therefore necessary to encode a significant amount of implicit knowledge into applications to make use of the data. To search in an existing repository, or to build an own tool on top of it, researchers need to be aware of, and understand this implicit semantics.

The SPARQL query language allows one to query explicitly for relations among resources. Such queries are impossible in the relational and in the object-oriented paradigm unless relationships are explicitly mapped to tables or, in the case of object-orientation, modeled as association classes. The latter, however, can make them difficult to distinguish from "real" classes. Given the high importance of relationships in software evolution, it is preferable to model them as first class objects—which is exactly what the Semantic Web does.

The importance of this aspect is emphasized in Section 3.5.3. There we introduce our recommender tool, which depends on the explicit semantics of ontologies. Given a set of data, it searches for certain types of individuals, as well as for their relations, to recommend appropriate visualizations.

3.3.4 Linked Software Evolution Data

With only relational database technology, synergies between research tools are hard to exploit. For example, we cannot simply establish connections between data stored in two different software repositories, such as a version control system and an issue tracker. The reason for this is that it is impossible to set a link from one repository to another—relations are local, not universal. Cross-domain queries spanning multiple repositories are impossible.

One of the driving forces behind the Semantic Web is the basic assumption that data becomes more useful the more it is interlinked with other data. The simple but powerful concept of statements represented by triples of URIs can be used to build an internet-scale graph of information because it makes it possible to link and query data that is stored in different locations.

The software analysis services described in Section 3.5.1 manage data based on these principles. URIs are assigned to every artifact analyzed and all the results generated. These URIs are de-referenceable over the Web and allow services to request from other (remote) services information about resources on an as-needed basis. Like that, the software analysis services already operate on a global graph of software evolution data today.

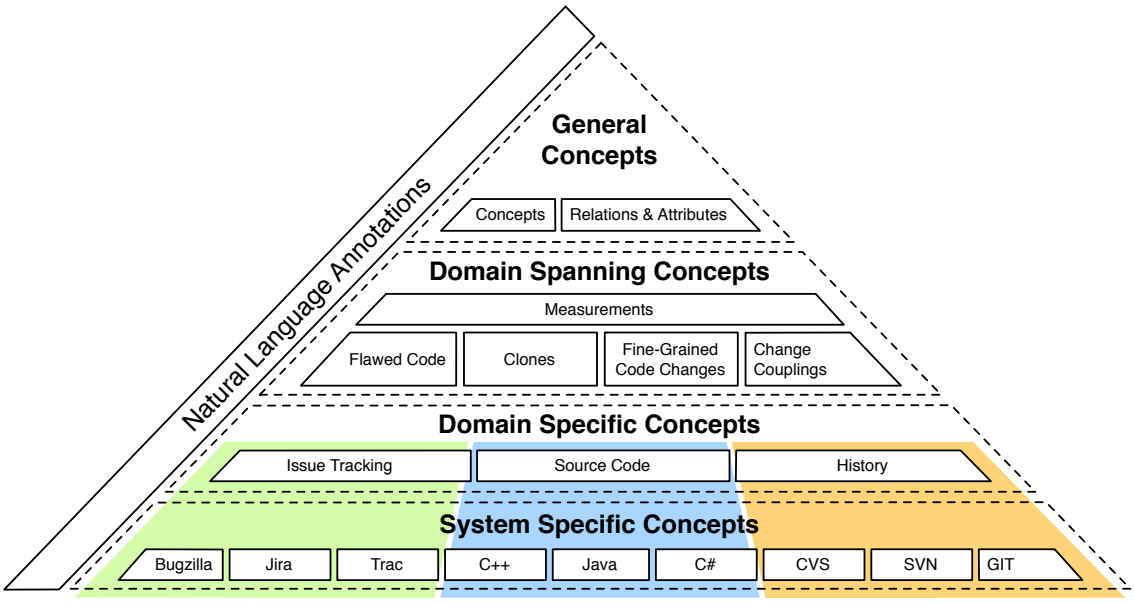


Figure 3.1: The Software Evolution Ontology Pyramid

In the next section, we describe SEON, an ontological description of the domain of software evolution. It exploits the characteristics of the Semantic Web mentioned above to support a wide range of semantics-aware applications.

3.4 SEON – A Pyramid of Ontologies for Software Evolution

The acronym SEON stands for *Software Evolution ONtologies* and represents our attempt to formally describe knowledge from the domain of software evolution analysis & mining software repositories. However, in contrast to many other existing ontologies, we did not aim to capture as much of the domain under discourse as possible. Instead, we originally incorporated only a limited set of concrete concepts and extended the ontologies solely when it was actually required by a particular analysis or by a tool that we had already built or used. Three of these tools are detailed in Section 3.5. We then followed a bottom-up approach and, from these very concrete concepts, iteratively added abstractions and extended our ontologies. This process is briefly described in Section 3.4.6.

Figure 3.1 presents an overview of the different layers of SEON. The most distinguishing feature is, compared to other ontologies related to the domain of software evolution, the strict organization into different levels of abstraction. In the following, we explain each of the layers that comprise our pyramid of ontologies. We focus on a few examples but do not provide a detailed description for every concept defined in SEON in this paper. Instead, we explain the general structure of our ontology pyramid and the rationale behind its design. Interested readers are invited to browse our OWL definitions online.[6] At the end of this section, we give an example on how the different layers can be used in conjunction with each other to describe knowledge in a concrete analysis scenario, namely the analysis of the evolution of code clones in a software system.

3.4.1 General Concepts

The pyramidion, *i.e.*, the top layer, is comprised of domain-independent or general concepts, the attributes that describe them, and the relations between the concepts.

Concepts are modeled by OWL classes. Instances of classes are OWL individuals. OWL datatype properties represent attributes, and OWL object properties the relations between concepts. The first ones link individuals to data values, whereas the latter ones link individuals to individuals. To better differentiate terms, we underline OWL classes in this section. A dotted underline denotes individuals and a dashed underline is used for properties.

Classes in the top-layer relate to concepts omnipresent in software evolution. Examples are Activity, Stakeholder, or File. We also defined a set of datatype properties for generic attributes, such as hasSize or createdOn. They are domain-independent; files, program execution stack traces, but also project teams have a size. Similarly, requirement documents, bug reports, or mailing list entries are attributed a creation date.

SEON also defines a more extensive set of domain-independent object properties. These properties are fundamental to many applications, as relations between "things" are paramount for most analyses in software evolution. On this level of abstraction, there is for example the concept of authorship, as any artifact in software evolution has one or several authors, denoted by the object property hasAuthor. Our ontology also has an object property called dependsOn that

[6] http://www.se-on.org/ontologies/

generalizes many different relations in the software evolution domain.[7] Specializations of dependsOn therefore can range from other domain-independent properties, such as hierarchical relationships (*i.e.*, a *parent-child* relationship), to more domain-specific ones, *e.g.*, dependencies between requirements or static source code dependencies. Such domain-specific properties, however, are specified in lower layers of SEON, as sub-properties of higher-level ones. Another domain-independent object property defines the abstract notion of similarity between two individuals. The concept of similarity, again, is universal. It applies to source code (a.k.a. "code clones"), as well as to issues (a.k.a. "bug duplicates") and many other artifacts. What "similar" actually means in a specific case, however, is then up to the fact extractors to decide when they instantiate SEON models.

What is the benefit of having defined the abstractions described above? First, we as human beings are comfortable with thinking in categories—this capability develops as early as within the first half year of our lives [HS04]. Categorization and taxonomizing things help us to understand the complex domain of software evolution. Second, as we will describe in the remainder of this paper, such abstractions enable us to build flexible, largely domain-independent tools to support many different facets of software evolution activities.

3.4.2 Domain-spanning Concepts

The second-highest layer of SEON defines domain-spanning concepts. These concepts are less abstract than the general concepts. They describe knowledge that spans a limited number of subdomains, *e.g.*, version control systems and source code in the case of our change coupling ontology. Change couplings describe implicit relationships between two or more software artifacts that frequently change together during evolution [BKPS97, GHJ98]. Other ontologies related to the version history of program code cover fine-grained source code changes and code clones. The ontology for fine-grained source code changes describes program modifications not only on a file level but also down to the statement level. It is based on the CHANGEDISTILLER meta-model of change types [FWPG07]. The code clone ontology is able to describe duplicated code and how it evolves over time.

[7] The concept of inheritance in OWL goes further than in object-orientation. Not only OWL classes can inherit from other classes, but also OWL object and data properties can inherit from other properties.

Similarly to the code clone ontology, our ontology about flawed code is concerned with quality attributes of source code. The ontology represents knowledge distilled from issue trackers and version control systems. It describes the bug history of files or modules, but also of individual classes or even methods in object-oriented programs. Furthermore, it covers *Design Disharmonies* [LMD05] or, in other words, formalized design shortcomings found in source code, *e.g.*, Brain Classes, Feature Envy, Shotgun Surgery, etc.

Another important concept is that of a Measurement. A sophisticated ontology for software measurement has been presented by Bertoa *et al.* in [BVG06]. SEON adapts some of the most important concepts identified by these authors, but we weigh simplicity over completeness by leaving out those that have not played a crucial role in our recent analyses.

A measurement is the act of measuring certain attributes of a software artifact or process; a Measure, or metric, is the approach taken to perform a measurement. Measures have a Unit, such as *number of bugs per line of code*. Measured values are expressed on a Scale, *e.g.*, an ordinal or nominal scale. Information about units and scales can be used to perform conversions, for example, to compare the results of different measurements. While the abstract concepts are defined in the pyramidion, many primitive measures are domain-specific. Still we consider measurements to belong mainly to the layer of domain-spanning concepts. Primitive measures, such as *number of lines of code* and *number of closed bugs*, on their own are not very meaningful and need to be put into relation in order to derive a meaningful assessment of a software system's health state. The most effective measurements therefore are based on derived measures [LMD05]; they present an aggregation of values from different subdomains. The *number of bugs per class* is computed from values originating from the source code and the issue tracker, and the *level of class ownership* is derived from source code and commits to a version control system.

In summary, SEON's layer of domain-spanning concepts describes software evolution knowledge on the level of analyses and results, whereas the remaining two layers describe raw data, *i.e.*, artifacts and meta-data directly retrieved from repositories.

3.4.3 Domain-specific Concepts

The third layer is divided into different domains corresponding to important facets of the software evolution process, that is, among others, issue and version management. It includes a taxonomy for source code artifacts encountered in object-oriented programming. While the concepts defined in this layer are specific to a domain, they are independent of technology, vendor, and version. Each domain captures the commonalities shared among the many different issue trackers, object-oriented programming languages, or version control systems.

The majority of issue trackers are organized around Issues that can be divided into Bugs, FeatureRequests, and Improvements. Issues are reportedBy someone and assigned to a developer for fixing them. Object-oriented programming languages usually consist of Classes organized in some kind of Namespaces. Classes declare Members—Methods and Fields—and they can inherit from other classes. Developers modify files in resolving issues and commit them to a version control system resulting in a new Revision for these files. They organize their repository with respect to development streams into Branches and prepare from time to time a Release of the system under development. All these concepts—and many more—are formally defined in SEON. These definitions build a taxonomy that can be shared among researchers and practitioners, but also among machines.

Concepts do not necessarily need to be present in all of the systems that are abstracted by the domain-specific layer. The concept of, *e.g.*, Mixins does not exist in Java but in other languages, such as Scala and Smalltalk. Defining this concept nonetheless is perfectly valid, as it is a common concept in object orientation. There will simply be no instances of such concepts if SEON is used to describe a software system written in Java or any other language that does not support them.

While devising the layer of domain-specific concepts, we maintained a bird's-eye view on commonly used technologies that are conceptually related, yet very different in implementation. Our goal was to distill some of the essentials of software evolution into a set of meta-models. These meta-models, however, are not static. They are destined to evolve, as the body of software engineering knowledge grows.

3.4.4 System-specific Concepts

Whereas the third layer describes domain-specific concepts that apply to families of systems, the bottom layer defines system-specific concepts. It extends the knowledge of the upper layers by concepts unique to certain programming languages, vendors, versions, or specific tool implementations. We aim to keep this layer as thin as possible while capturing relevant information beneficial for analyzing specific facets of the evolution of concrete programs. For some systems, we have barely seen the need to define specific concepts, without loosing crucial information. Other systems differ significantly from the baseline and require more system-specific knowledge.

One example for system-specifics is the severity of issues. While most modern issue trackers know the concept of severity to classify an issue, their concrete implementations vary quite substantially. The different levels of severity, as well as their naming, depends very much on the particular issue tracker and, in some cases, even on how it is configured by development teams. Still, the information is valuable, *e.g.*, as input for machine learning algorithms when experimenting with automated bug triaging approaches [GPG10]. Therefore we defined Severity in the layer of domain-specific concepts, but the individuals that represent the different levels of severity are covered in system-specific ontologies. System-specific parsers then extract this information and link individuals of Issue to the corresponding individuals of Severity.

3.4.5 Natural Language Annotations

The Semantic Web was not primarily devised for human beings consuming information. Instead its conception is that machines become capable of processing the knowledge of humans and there is usually additional effort of knowledge engineers needed to encode it in an adequate format.

Despite this machine-centric design, there are many occasions where humans need to interface with Semantic Web data. Therefore, we added a layer of natural language annotations to SEON. These annotations provide human-readable labels for all classes and properties. For individuals, we use RDF Schema labels (*rdfs:label*).

In particular, we defined the following custom annotations as subclasses of the OWL *AnnotationProperty*.[8] The most important three annotations in the natural language layer are:

- **phrase-s** adds singular synonyms to OWL classes and properties.
- **phrase-p** adds plural synonyms to OWL classes and properties.
- **explanation** adds a human-readable description to OWL classes or properties

The encoding of the grammatical number of a synonym (*phrase-s* vs. *phrase-p* annotation) is important in order to correctly translate statements from OWL to natural language. The *explanation* annotation is very similar to the RDF Schema comment annotation (*rdfs:comment*) defined by the W3C, except that our annotation is explicitly meant to be shown in user interfaces to end-users (*e.g.*, in tooltips), whereas *rdfs:comment* is also often used to document OWL classes and properties for knowledge engineers.

In Figure 3.2, we show an excerpt of an RDF graph as an example of how we annotate our SEON ontologies with natural language. For the concept Developer, we added multiple natural language representations, in particular the nouns *Author(s)*, *Developer(s)*, and *Programmer(s)*. The annotations from its super-concept Stakeholder—*Stakeholder(s)*, *Person(s)*, and *People*—also apply to Developer. Same applies for properties, where for example changes is annotated with the verbs *change(s)*, *modify*, *modifies*, and *edit(s)*.

[8] `http://www.w3.org/2002/07/owl#AnnotationProperty`

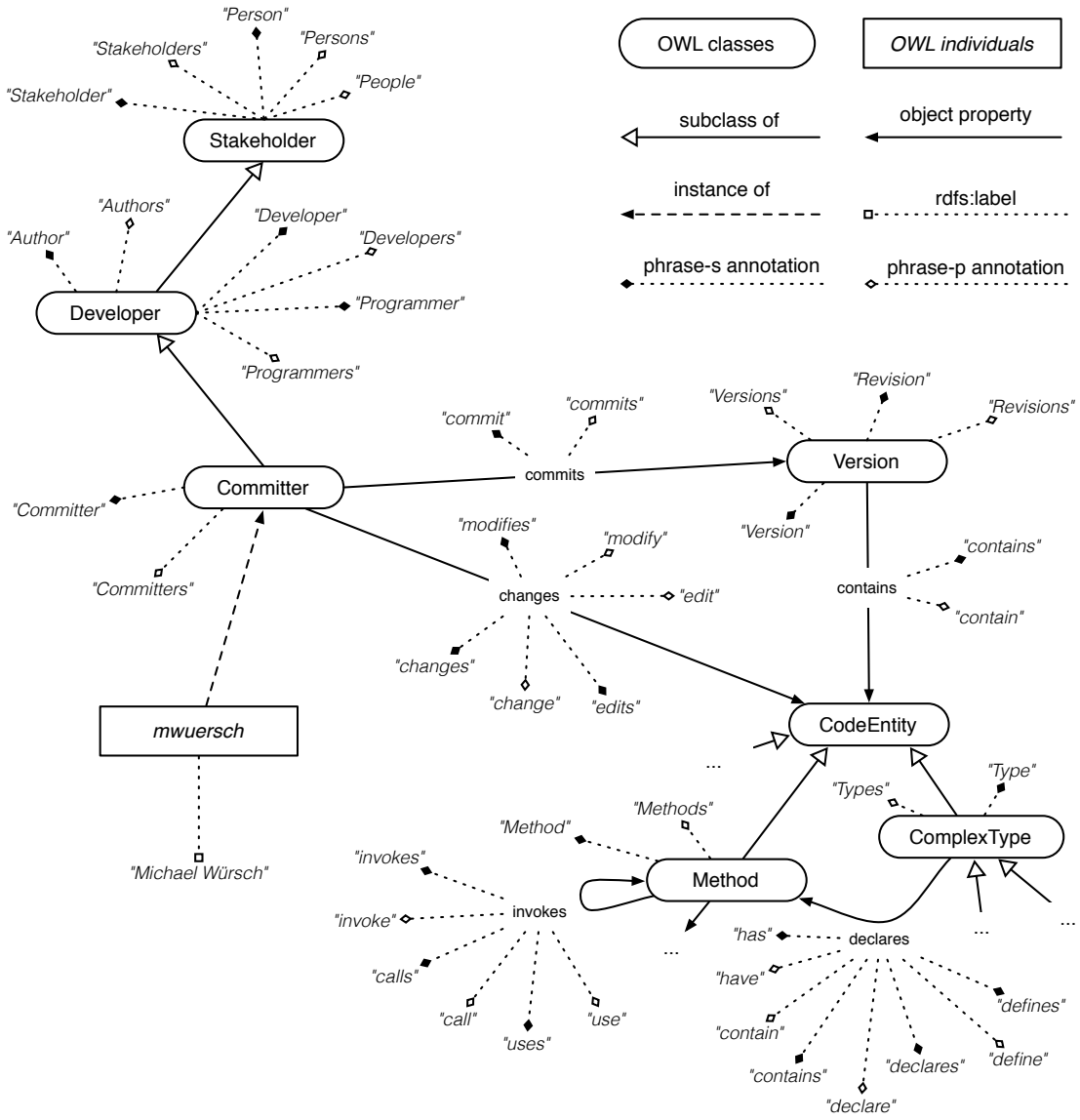


Figure 3.2: RDF Graph with Natural Language Annotations

In contrast to OWL classes and properties, where the annotations are encoded directly in SEON, fact extractors have to generate meaningful *rdfs:label* values for individuals. In most cases, this process is straightforward: for Java classes, fields, and methods, the Java identifier is taken, whereas for bug reports, the issue-key provided by the issue tracker (*e.g.*, "IVY-123" for the issue #123 of the Apache Ivy project) serves as label.

Both, the annotations and *rdfs:labels* are key to the query approach that we

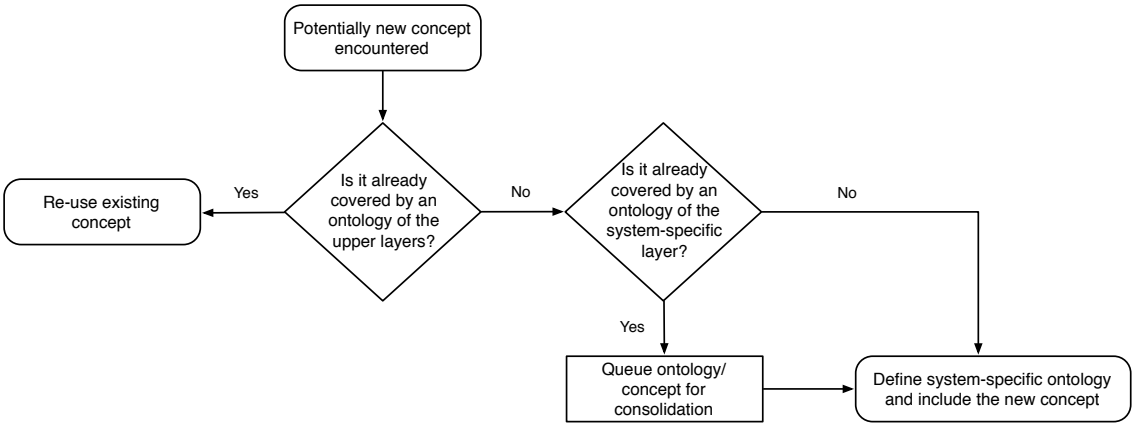


Figure 3.3: Informal Design Process when encountering concepts during the conception of an analysis

discuss in Section 3.5.2. When entering queries, the nouns and verbs are used to provide guidance in composing questions, such as *"Which Programmer modifies the method foo()?"* or *"What methods call bar()?"*. The natural language annotations of SEON also enhance some of the Web front-ends of the software analysis services presented in Section 3.5.1. The annotations are used to automate the generation of simple human-readable reports, *e.g.*, "Michael Würsch commits Revisions 1-100." or "The class DBAccess has changed 50 times.".

3.4.6 Our Knowledge Engineering Process

Choosing which concepts should be included in an ontology in general, and assigning concepts to a layer of SEON in particular, is not always straight-forward. In the following we therefore briefly sketch the informal ontology design process used for SEON, which is illustrated in Figure 3.3.

Knowledge engineers often start from an abstract high-level view when they identify and describe the important concepts in a domain under discourse. Then these concepts are iteratively validated and refined against the reality. In contrast to this top-down approach, we follow a more data-driven, bottom-up approach. At the beginning of the conception phase of a new software evolution support tool or data importer, we quickly model the important concepts of its domain, while neglecting those concepts that are not of immediate use for our purpose. For each important concept, we check whether it is already represented in one of

SEON upper layers, *e.g.*, the domain-specific layer, and re-use the existing concepts whenever possible. If the concept is not yet defined, we first stage the concept in a system-specific ontology for the specific system. Additionally, we check whether we have already defined similar concepts in other system-specific ontologies and, if so, queue them for consolidation. We usually post-pone the consolidation step until we reached a sufficient understanding of the problem domain—system-specific ontologies therefore act like an incubator to new concepts.

When we model, for example, the concepts of the two programming languages Java and C++, we first create two distinct system-specific ontologies. Then we compare the results and move the commonalities, such as Class, Field, Method, extends, invokes, etc., to SEON's domain-specific layer. The concepts that apply only to C++, such as Struct, Function Pointer, Header File, and the Java-exclusive concepts, *e.g.*, Interface, Annotation, and Inner (Anonymous) Class, remain in the respective system-specific ontologies. Pervasive concepts, *i.e.*, those that apply to multiple domains, for example File, are promoted from the domain-specific to the domain-spanning—or even to the general layer of SEON.

3.4.7 An Example Scenario: Clone Evolution

Code clone detection in source code has been a lively field of research for many years now and it is generally accepted that duplicated code violates the *Don't Repeat Yourself (DRY)* principle [HT99], which can lead to software that is harder to maintain. An interesting aspect of code duplication is how clones evolve over time. This was, for example, investigated by Kim *et al.* in [KSNM05].

Now consider the following scenario, where a researcher decides to carry out a similar study to the one presented by Kim *et al.* In particular, the researcher wants to find out whether the number and size of duplicated fragments change over the lifetime of a Java program. We assume that a clone detector was already selected; scripts to check-out every version of the source code files from an SVN repository have been developed. What is left, is to devise a tool that runs the clone detector on the data to perform the analysis. For that, the researcher needs to decide what meta-model should be used to represent the data under analysis, as well as the results of the analysis.

SEON provides all the necessary means to describe such knowledge. In the following, we briefly discuss how the relevant concepts and their relations are

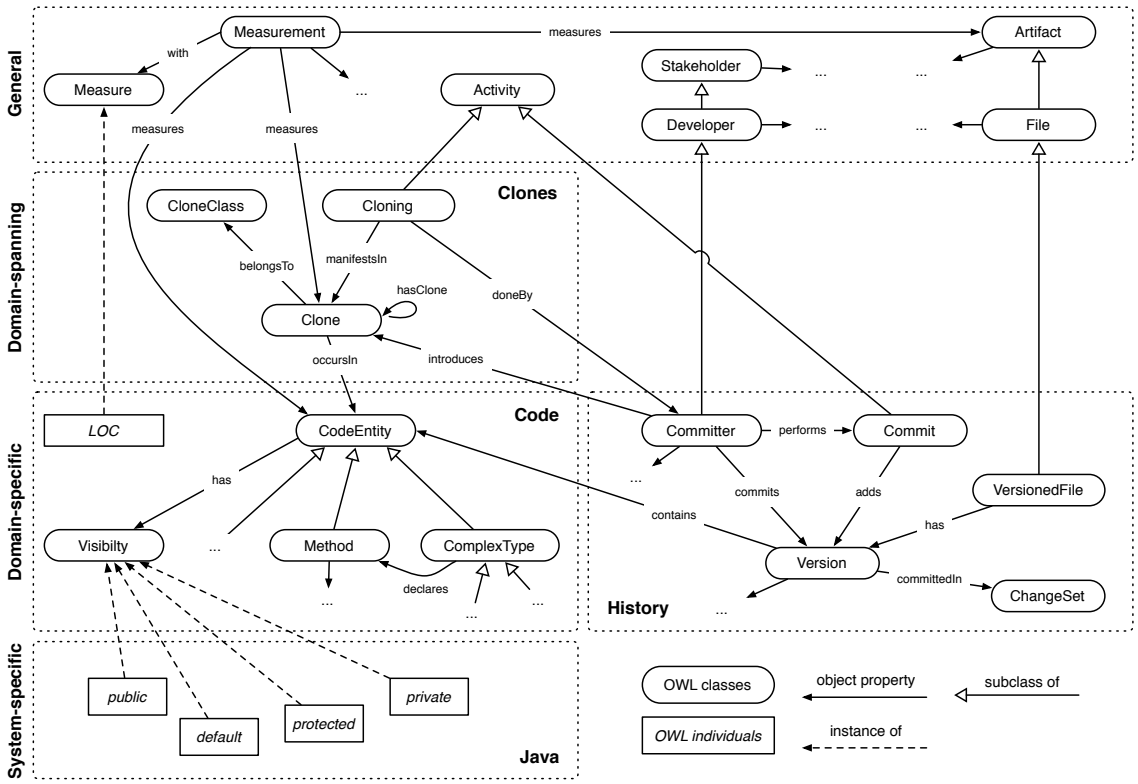


Figure 3.4: The SEON concepts involved in a Clone Evolution Analysis Scenario

distributed over the four layers of our ontology pyramid. The OWL classes and object properties for the scenario are illustrated in Figure 3.4. The illustration omits datatype properties for the sake of simplicity.

The core concept for this analysis is Clone. A clone belongsTo a CloneClass of duplicated fragments that are similar in syntax or semantics. While the concepts of our clone ontology might not suffice to represent all possible variants of clone analyses, it is straightforward to extend the existing ones. For example, one could specialize the concept Clone with different types of clones, such as SemanticClone or SyntacticClone to provide further classification. Or, additional object properties could link clones to issues for investigations on whether duplication leads to more bugs, and so on.

A Committer introduces a clone when she commits a new Version of a VersionedFile to the SVN repository. Committers are Developers that can check-in modifications. They are one of the many Stakeholders associated with the

```
Cloning(?cloning), doneBy(?cloning, ?committer),
manifestsIn(?cloning, ?clone) → introduces(?committer, ?clone)
```

Listing 3.1: An Example for a SWRL rule defined by SEON

development process. Versioned files are Files managed by a version control system. Files are among the Artifacts that are produced when software is created. Clones occur in a particular CodeEntity, such as in a ComplexType (*i.e.*, a class, interface, enum, etc.), a Method, etc. The size of such a piece of code, as well as the size of a clone, can be assessed by a Measurement. An adequate Measure for that is the number of lines of code, LOC.

The OWL classes Cloning and Commit are special cases: in principle, the relationship between clones and committers is already sufficiently stated by the object property introduces. However, in some cases, we also want to express that the introduction of a clone is an Activity with a certain time stamp and carried out by a particular stakeholder. There are two ways to do that. The first is reification, which allows for statements about statements. The second is to define an association class. Since reification has not been widely adopted in the Semantic Web, we decided for the second variant and defined the OWL class Cloning to represent the introduction of a clone. A clone introduction is doneBy a committer and manifestsIn a new clone. A similar case is that of a Commit. It is also an activity that a committer performs and which adds a new version to a file. This apparent redundancy in the ontology definition allows us to support a wider range of applications. The query approach discussed in Section 3.5.2 works better with triples, such as *"Committer$_A$ commits Version$_B$"*, that are close to the subject-predicate-objected sentence structure in English. On the other hand, the tool presented in Section 3.5.3 explicitly queries for activities to generate, *e.g.*, timeline views. Fact extractors do not necessarily need to create both, an individual of Cloning and the statement *"Committer$_X$ introduces Clone$_Y$"*. In many cases, we defined rules in the Semantic Web Rule Language (SWRL) [HPSB+04], similar to the one in Listing 3.1. The rule states that, if some cloning activity has been carried out by a committer, and the cloning manifested in a clone, then the committer has introduced a new clone. With a reasoner, we can then automatically infer the missing triples for particular cases.

Notable in Figure 3.4 is also the OWL class Visibility. In most object-oriented

```
SELECT ?clone ?size ?version
WHERE
   {  ?code          rdf:type        seon:CodeEntity
      ?clone         rdf:type        seon:Clone ;
                     seon:occursIn   ?code
      ?version       rdf:type        seon:Version ;
                     seon:contains   ?code
      ?measurement   rdf:type        seon:Measurement ;
                     seon:with       seon:LOC ;
                     seon:hasValue   ?size ;
                     seon:measures   ?clone }
```

Listing 3.2: SPARQL query returning Clones incl. size and version they appear in

programming languages, there exists an information-hiding mechanism to control the access of parts of the code. In Java, there are the visibility modifiers public, default, protected, and private that apply to types and their members. The actual instances of the visibility modifiers are defined in a system-specific (**Java**) ontology because there are quite significant differences in the meaning of such modifiers depending on the programming language used. The visibility concept, however, belongs to the domain-specific layer together with the other abstractions of **Code**. The layer also contains the predefined LOC individual, because the measure is clearly associated with program code. In our analysis scenario, there are no domain-spanning measures needed. The **History** ontology is located at the same level of abstraction as the **Code**. Currently, there are no system-specific extensions to it. The **Clones** ontology is domain-spanning—it relates to the **Code**, as well as to the **History**. The general concepts layer then provides abstractions for various concepts used in the lower layers.

Coming back to our initial clone evolution analysis scenario, we conclude that SEON provides the concepts necessary to support it. Once the ontology has been populated by a fact extractor, a concise SPARQL query can be issued to retrieve all clones, their size, and the versions they occur in. The query is given in Listing 3.2. Note that we have left out the prefix definition part of the query: the prefix *rdf* refers to `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, whereas we assume that the *seon* prefix stands for `http://se-on.org/ontologies/`. In reality, each of the different layers of SEON has its own prefix/namespace.

3.5 Applications powered by SEON

In the following, we describe three different applications that work with SEON as their semantic backbone. The first one is our software evolution analysis web service platform SOFAS; the second one is HAWKSHAW, a natural language interface for answering program comprehension questions; and the third application is a recommender system called Semantic Visualization Broker (SVB). SVB analyzes the semantics of a given set of data and comes up with a list of visualizations that could be helpful to gain a deeper understanding of the software system under analysis. We have fully implemented the three approaches in proof-of-concept tools. SOFAS and HAWKSHAW are even available for download on the SEON website.

3.5.1 Software Analysis Services

Mining Software Repositories has been an active field of research for many years, and various analysis techniques have been proposed, based on the idea that software engineers can learn from the development history of programs.

No matter whether these approaches are concerned with code analysis, code duplication, bug prediction, or any of the other repository-based analyses, many of them have in common that researchers had to build data extractors for version control repositories, issue trackers, mailing lists, and so on. While these efforts share many similarities, synergies are hard to exploit as many tools were designed to work stand-alone. The outcome is a diversity of platforms, similar, yet incompatible meta-models, and tool-specific input and output formats.

To overcome these challenges, we have devised SOFAS[9] (SOFtware Analysis Services), which we presented in [GG11]. SOFAS allows for a simple yet effective provisioning and use of software analyses based upon the principles of Representational State Transfer (REST, as introduced by Fielding in [Fie00]) around resources on the Web.

An overview on the architecture of SOFAS is given in Figure 3.5. The architecture is made up by three main constituents: *Software Analysis Web Services*, a *Software Analysis Broker*, and *Software Analysis Ontologies* being part of SEON. The

[9] SOFAS is available online at `http://se-on.org/sofas/`

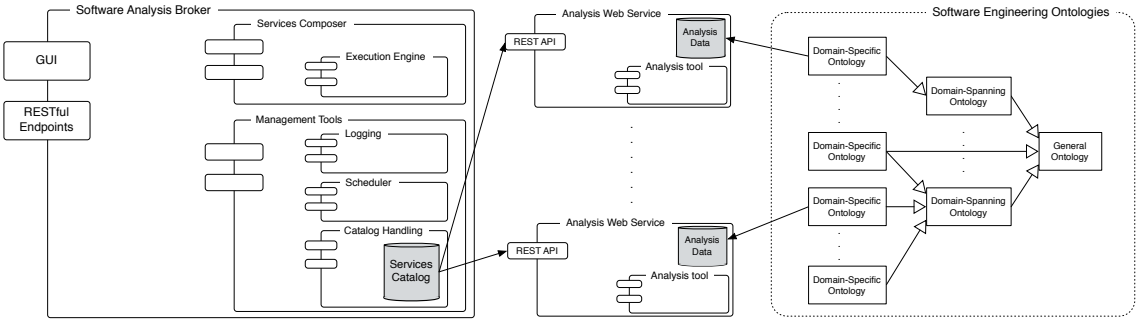


Figure 3.5: The SOFAS Architecture ([GG11])

software analysis web services "wrap" already existing analysis tools by exposing their functionalities and data through standard RESTful web service interfaces. The broker acts as the services manager and the interface between the services and the users. It contains a catalog of all the registered analysis services with respect to a specific software analysis taxonomy. As such, the domain of analysis services is described in a semantical way enabling users to browse and search for their analysis service of interest. SEON defines and represents the data consumed and produced by the different services.

REST provides us a truly uniform interface to describe all the analysis services in the SOFAS architecture, the structure of their input and output, and how to invoke them at a syntactic level. However, there is no way to programmatically know what a service actually offers and what the data means that it consumes and produces. Ontologies in general, and SEON in particular, help tackling both problems by providing meaningful service descriptions and data representation.

The Semantic Web leverages SOFAS in multiple ways. First, every resource gets a de-referenceable URI assigned. URIs align well with the REST principles and allow one service to hand-over artifacts to another one in a straight-forward manner. Next, the formal data semantics achieved with SEON helps in clearly specifying the input expected, as well as the output generated by the services, which increases interoperability and simplifies reuse of processing results. This is achieved by slightly expanding the Web Application Description Language (WADL) [Had09] with annotations inspired by SAWSDL (Semantic Annotations for WSDL) [FL07]. With them, the input and output of the services can be declared as being described by SEON. Last but not least, the footprint of the information exchanged by the services can be reduced by incorporating a reasoner. Only a

limited set of triples then needs to be passed along by the sender and reasoning can be done by the receiver to add additional triples, if needed.

3.5.2 Supporting Developers with Natural Language

In [WGRG10] we presented a framework for software engineers to answer common program comprehension questions with *guided-input natural language* queries, for example those questions presented by Silito *et al.* in [SMDV06]. The framework is called HAWKSHAW[10] and has been implemented as a set of plug-ins for the Eclipse IDE. Figure 3.6 shows a screenshot of our query interface in action. In the example, a user has already started to compose a query. Three words have been typed in so far, *"What Method invokes"*, and the drop-down menu presents the full list of methods that can be entered to complete the query.

The HAWKSHAW approach follows a method coined *Conceptual Authoring* or WYSIWYM (What You See Is What You Meant) by Hallet *et al.* in [HSP07] and Power *et al.* in [PSE98]. This means that, for composing queries, all editing operations are defined directly on an underlying logical representation, in our case SEON. However, the users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology.

We use a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded [BKKK06]. The static part needs to be defined manually and, additionally, contains information on how to translate the user input into SPARQL. We generate the dynamic part from labels of the individuals, from the identifiers of the classes and properties, as well as from the SEON natural language annotations (see Section 3.4).

The static grammar basically defines a stub. In the example given above, the grammar describes that, after one of the interrogative determiners "What" or "Which", the subject of the sentence needs to follow. The subject needs to be an OWL class defined by SEON. Further, the verb of the sentence has to be an object property that fits the subject, *i.e.*, the object property has the class in its domain that has been selected as the sentence's subject. Object properties not fulfilling this

[10] Our tool is named after Hawkshaw the Detective, a comic strip popular in the first half of the 20th century. *Hawkshaw* meant a detective in the slang of that time. The tool HAWKSHAW is available for download at `http://se-on.org/hawkshaw/`

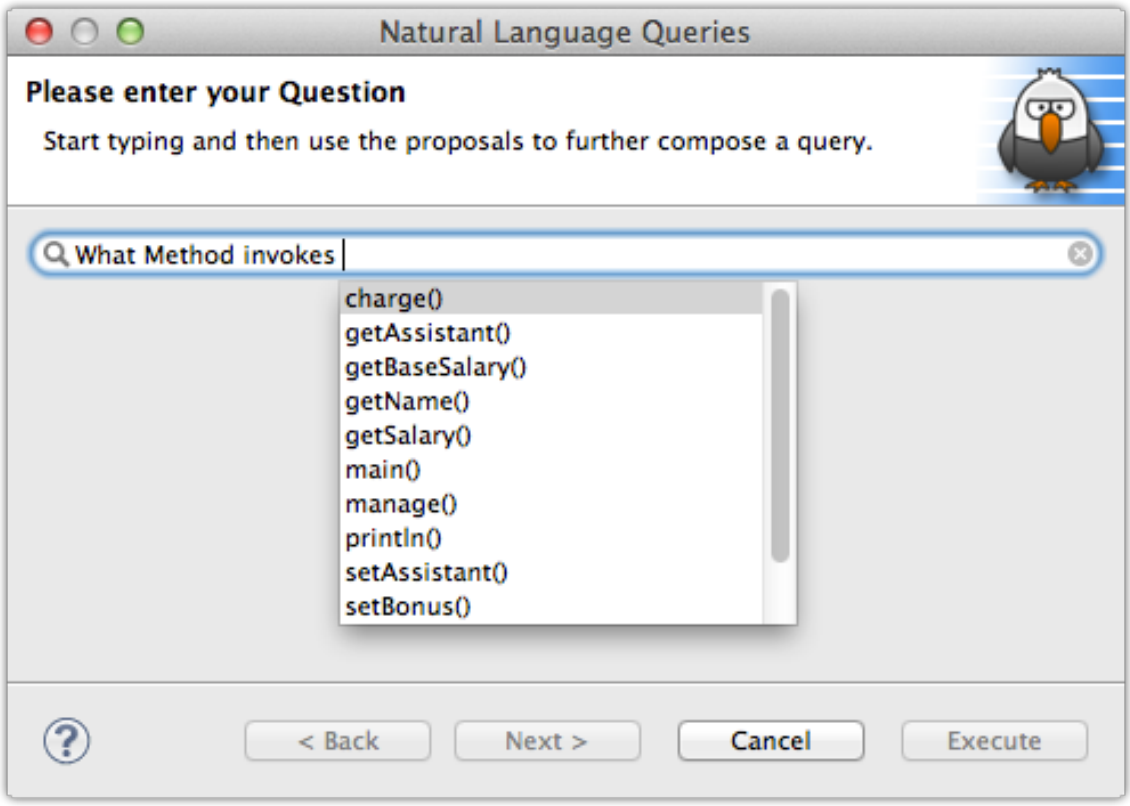


Figure 3.6: The Guided-Input Natural Language Interface powered by SEON

constraint will not be presented to the user. Similarly the object of the sentence is an individual of a class in the ontology. The individual's class has to comply to the range specified for the object property, otherwise it will not be shown either. The stub provided by the static grammar then looks as follows: "What <class> <object-property> <individual> ?"

The dynamic part of the grammar provides the replacements for the placeholders in the stub (denoted by < >). These replacements are presented to the user. Consider *"What Method*[1] *invokes*[2] *charge()*[3] *?"*. In this query, ([1]) is a label for the OWL class *JavaMethod*, ([2]) comes from the object property *invokesMethod*, and ([3]) from a human-readable label for one of the OWL individuals that have the class *JavaMethod*.

The utilization of the SEON ontologies for driving HAWKSHAW yields several major benefits: Ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles how humans talk about things and can be

easily transformed into natural language sentences. A surprisingly small set of static grammar rules allows for a variety of different queries.

Properties in OWL are a binary relation that can be restricted by specifying *domain* and *range*. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, this information can be exploited to filter the verbs that can follow a given subject, or the objects that can follow a given verb. For example the question *"Which developer is assigned to issue #133?"* makes sense, whereas *"What field invokes class A?"* does not.

We employ the Pellet reasoner [SPG+07] to infer specializations or generalizations. When we ask for, *e.g.*, *"What persons are contributing to project X?"*, we are not only interested in a list of direct instances of the concept *Person*, but also in specializations, such as *Developers*, *Testers*, etc. Similarly, whenever we know that developers *create* or *change* an artifact, we also want to generalize that they are *contributing* to the project. Reasoners greatly simplify data extraction, as they reduce the amount of explicit information that we need to state in our models.

3.5.3 Semantic Visualization Broker

The third application presented in this paper addresses the hardly known capabilities of software visualizations. The Semantic Visualization Broker (SVB) is essentially a recommender tool that suggests to the user suitable visualizations for a given set of data. The data can originate from the results of a query composed with the HAWKSHAW approach (Section 3.5.2), but also from a SOFAS analysis workflow (Section 3.5.1), or virtually any other source of RDF/OWL data.

Visualization plug-ins can register themselves with the SVB and specify the semantics of the data they can handle. The SVB expects as input a knowledge base and a result set. The result set should consist of the information a user asked for, whereas the knowledge base provides the context, in case that the SVB or a visualization has to query for additional data. The SVB then invokes a reasoner to infer abstractions from the result set and compares the outcome with the registration that the visualization plug-ins have provided. Any matches are presented to the user. The user can then select one or several recommendations from the list and the SVB will invoke and configure the visualizations automatically with the input data.

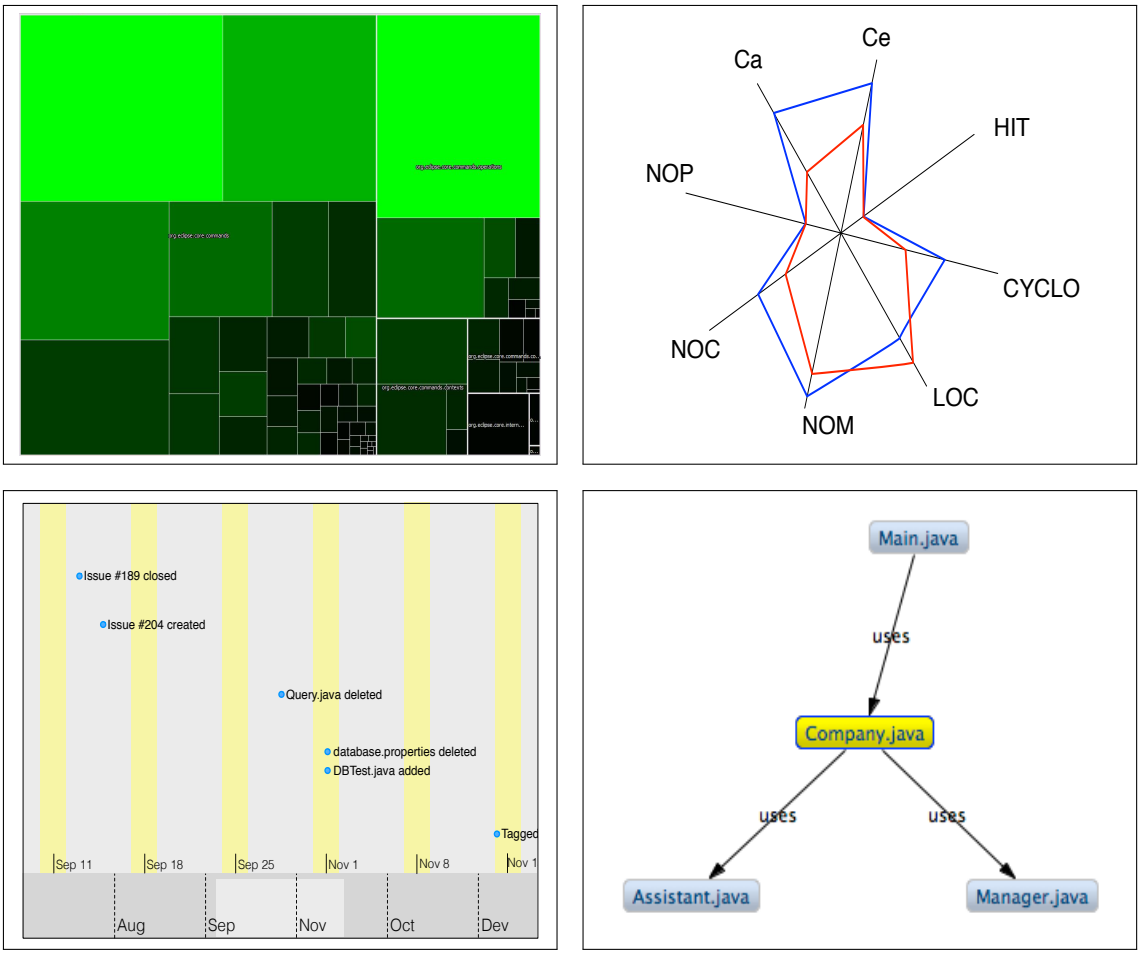


Figure 3.7: The types of visualizations currently supported by the Semantic Visualization Broker: The upper left figure shows a tree map of a Java system, the upper right one shows a radar chart with measurements for two different versions of a Java class, the lower left figure shows a timeline with software evolution activities, and the lower right one shows a simple graph-based explorer displaying the dependencies among four Java classes.

When the SVB receives a set of individuals as result set, it will query the knowledge base for their data properties and for object properties that link those individuals together. We currently support four different scenarios, which we describe in the following. An overview on the implemented visualization types is given in Figure 3.7.

Hierarchies. If the SVB detects a hierarchical relationship between the individuals in the result set, it will recommend a simple tree-like widget (which has been omitted from Figure 3.7—it is similar to the widgets well-known from file system explorers) and a tree map visualization. If the selected individuals have a size measurement assigned (*e.g.*, for files the *lines of code* metric), the SVB will configure the tree map to incorporate the size of each individual to calculate the layout.

Measurements. If more than one individual has measurements assigned, then the SVB recommends a visualization based on Radar Charts. Each axis of the chart represents a certain type of measure. The number of axes that are displayed is limited; whenever more measures are available, some of them are chosen randomly and the user is given the possibility to reconfigure the selection. If measurements are available for more than one version of the individuals (*e.g.*, for files under version control), then each axis will display multiple entries.

Activities. In the case that most of the individuals represent an activity with a timestamp assigned, the SVB will automatically come up with a scrollable timeline-like visualization.

Miscellaneous data. As a fallback, if none of the cases above apply, the broker will suggest a simple graph-based explorer that displays individuals and data values as nodes and properties as edges. Unless the properties are defined as being *symmetric*, the corresponding edges will be directed.

Labels displayed in each of the visualizations are derived either from the RDF Schema labels or from the natural language annotations of SEON. The clear, machine-processable semantics of the data enable the SVB to make educated guesses on what visualizations may be appropriate. The power of a reasoner allows us to specify the concepts and relations supported by a visualization in a very generic way—the reasoner will automatically infer a hierarchical relationship from a set of triples containing, *"$Class_A$ declaresMethod $Method_B$"* and propose a tree-based visualization consequently.

The SVB offers quite some potential for enhancements. For example, we will explore the range of visualizations it can support and to what extent it is generalizable to non-visual applications.

3.6 Related Work

In this section, we briefly sketch existing work involving ontologies in software engineering. We refrain from discussing publications that are only related to the approaches presented in Section 3.5, but not particularly to the Semantic Web and ontologies. Related work in the context of software analysis services was already given in [GG11], whereas research in the area of program comprehension and developer support has been discussed extensively in [WGRG10].

A general overview of applications of ontologies in software engineering has been given in [GL02,HS06,UJ96]. All of these publications promoted the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. In the following, we elaborate on how ontologies were applied to advance particular fields of research in software engineering. To the best of our knowledge, SEON is the only approach that describes software evolution data on multiple abstraction layers. Another unique selling proposition of our family of ontologies is that they were validated in three very distinct scenarios (*cf.* Section 3.5), whereas most other ontologies were deployed only in a rather specific environment.

3.6.1 Ontologies for Software Artifacts

Different approaches to establish taxonomies for software engineering by means of ontologies have been presented recently.

Hyland-Wood *et al.* [HWCK06] proposed an OWL ontology of software engineering concepts, including classes, tests, metrics, and requirements. Bertoa *et al.* focused on software measurement [BVG06]. Their software measurement ontology influenced the respective concepts of SEON. Bertoa *et al.*'s set of measurement concepts is more complete, whereas our ontology focuses on simplicity.

Oberle *et al.* recognized that *the domain of software is a primary candidate for being formalized in an ontology* [OGS09], being both, sufficiently complex and reasonably stable in paradigms and aspects. Consequently, a reference ontology for software was presented to distinguish fundamental concepts in the domain of software engineering, such as data and software.

These three approaches show some overlap in concepts with our ontologies

but they neglected evolutionary aspects, whereas SEON explicitly models the development history of software systems, such as versions, releases, bugs, etc.

3.6.2 Ontologies for Software Maintenance

Several approaches relied on ontologies to support software maintenance—be it to describe domain knowledge of developers, source code and documentation to support program comprehension, or to infer bugs based on a set of heuristics.

LaSSIE, presented by Devanbu *et al.* in [DBS91], was an early attempt to integrate multiple views on a software system in a knowledge base. It also provided semantic retrieval through a natural language interface. Frame systems, a conceptual predecessor to the ontologies of the Semantic Web, were used to encode the knowledge. The main goal of LaSSIE was to preserve knowledge of the application domain for maintainers of the software system.

The author of [Wel97] found LaSSIE's source code model too course-grained and not applicable to object-oriented code. Therefore, he augmented abstract syntax trees with semantics. For that DL was used to develop an ontology for software understanding. The ontology, in combination with an inferencer, then enabled automatic detection of side effects in code and path-tracing.

Witte *et al.* [WZR07] used text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings were represented in RDF.

Yu *et al.* also represented static source code information by means of an OWL ontology [YZY+08]. They further used the Semantic Web Rule Language (SWRL) [HPSB+04] to describe common bugs found in code. With a rule engine, inference results could be obtained to indicate the presence of bugs.

Our natural language query approach HAWKSHAW described in Section 3.5.2 shares many similarities with LaSSIE but, thanks to SEON, potentially covers a broader range of concepts. However, SEON does not incorporate application-specific knowledge. The other three approaches described above focus only on source code, whereas we incorporate many different artifacts, stakeholders, and their activities.

3.6.3 Ontologies for Software Reuse

Properties of software components have been represented with ontologies in the past. Such properties ranged from programming languages and source code facts to licenses, software types and application domains. The common goal was to foster reuse by enabling searches in a component database for certain criteria that relate to, *e.g.*, particular requirements.

Happel *et al.* [HKST06] proposed various ontologies to foster software reuse. In their KOntoR approach, they provided background knowledge about software artifacts, such as the programming language used or licensing models. The artifacts, along with their ontology meta-data, were stored in a query-able central repository to facilitate reuse.

The authors of [HKF08] used ontologies to describe software components. They classified software with respect to a hierarchy of software types. An example given in their paper was IBM's DB2, which is a relational database management system (RDBMS); RDBMSs were then considered as a subclass of database managements systems, and so on. The authors additionally defined hierarchies of functionality types (*e.g.*, *importing data* as a special kind of *adding data*) to further describe the features of components. An algorithm was presented to automatically find an optimal component solution for a given set of requirements.

Dietrich *et al.* developed a tool that scans the abstract syntax tree of Java programs and detects design patterns for documentation purposes [DE05]. The design patterns were described in terms of OWL ontologies.

Alnusair and Zhao [AZ11], similar to Hartig *et al.*, used OWL ontologies for component descriptions. They took a three-layered approach for their ontological descriptions: an ontology representing static source code information, different domain ontologies to conceptualize the domain of each component (*e.g.*, finance or medicine), and an ontology that extended their source code ontology with component-related concepts. The authors supported several kinds of query methods against their component knowledge base: type or signature-based queries, meta-data keyword queries, or pure semantic-based queries.

SEON, in contrast to these four approaches, does neither model software systems at a component level, nor does it represent design patterns. However, in our ontologies, we model other important facets of software that could yield interesting synergies when synthesized with these ontologies for software reuse, for

example, to give insights on the maintainability of particular components. This could help software engineers to make even more profound decisions on what components their software systems should be based on.

3.6.4 Ontologies in Search-Driven Software Engineering

The field of search-driven software engineering has produced various code search engines. Some of them simply use OWL/RDF as an internal representation of program code and allow users to issue SPARQL queries against the code base [KRSR10]. Others exploit the possibilities of the Semantic Web further. Durão *et al.*, for example, classified source code according to domains, such as Graphical User Interfaces, I/O, Networking, Security, and so on [DaVAdLM08]. The authors then provided a keyword search over the code base, and the results of the queries could be limited to return only matches from a particular domain.

The applications of SEON presented in Section 3.5 also make extensive use of the Semantic Web's search facilities, in particular of SPARQL. Source code search, however, is not the main purpose of our applications but rather a means to an end. Nevertheless, it is easily conceivable that we might adopt a code search engine as a SOFAS service in the future.

3.6.5 Ontologies in Mining Software Repositories

Several researchers have described software evolution artifacts found in software repositories with OWL ontologies. Their approaches integrated different artifact sources to facilitate common repository mining activities. The flexible RDF data model, automatic semantic mashup technologies, and the powerful search-facilities of the Semantic Web have proven their use in this context.

Tappolet made a case for incorporating Semantic Web technology in software repositories in [Tap08]. The authors claimed that this would greatly facilitate the handling of distributed and heterogeneous software project data. Tappolet then presented a roadmap towards such semantics-aware software project repositories consisting of three main steps: 1) data representation by means of RDF/OWL ontologies, 2) intra-project repository integration, and finally 3) inter-project repository integration.

Based on these ideas, Kiefer *et al.* presented EvoOnt [KBT07], a software repository data exchange format based on OWL. EvoOnt involved three sub-ontologies: a software ontology model, a bug ontology model, and a version ontology model. The authors used a modified version of SPARQL to detect bad code smells, calculate metrics, and to extract data for visualizing changes in code over time. A reasoner was incorporated to detect orphan methods, *i.e.*, methods never called by any other methods in the system. Tappolet *et al.* recently extended the EvoOnt approach. Several software evolution analysis experiments from previous Mining Software Repositories Workshops were repeated and it was demonstrated by the authors that, if the data used for analysis were available in EvoOnt, then the analyses in 75% of the selected MSR papers could be reduced to one or at most two simple SPARQL queries.

Iqbal *et al.* discussed different scenarios and use cases for Linked Data in software engineering in [IUHT09]. They presented their *Linked Data Driven Software Development* (LD2SD) methodology, which involves transformation of software repository data into the RDF format and then indexing with a semantic indexer. The overall goal was to provide a uniform and central RDF-based access to JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc. Integration between the repositories was achieved with *Semantic Pipes*, an RDF-based mashup technology. The results were finally injected into the DOM of a Web page (*e.g.*, that of a bug tracker) to provide developers with context-related information.

None of these approaches organize their ontologies in consecutive layers of abstractions with clear representational purpose, as we did for SEON. Instead, the authors have laid out their ontologies at a particular level of abstraction. For example, while most concepts in EvoOnt can be mapped 1:1 to concepts in SEON, it is conceptually situated somewhere between SEON's system- and domain-specific layers and lacks the domain-spanning and general concepts that we have defined.

Despite these limitations, we can envision interesting interactions between our semantics-aware applications and the technologies presented by the other authors. The SPARQL extension presented by Kiefer *et al.*, for example, adds machine learning algorithms (SPARQL-ML [KBL08]) and similarity joins (iSPARQL [KBS07]) to the Semantic Web. Both extensions could lead to a complete new family of SOFAS services or at least simplify the implementation of existing ones. The semantic mashup technology used in LD2SD could further improve the presentation of the analysis results of our services.

3.7 Conclusions

Some decades ago, a team of developers could write industrial-strength software on their own, only with the aid of a simple text editor, a compiler, and perhaps a debugger. The software engineering landscape has changed dramatically since then.

Development teams have grown to dozens, and sometimes even hundreds of people. A plethora of tools have found their way into integrated development environments—without the help of these IDEs, we as programmers can barely imagine to write a single line of code anymore. Software repositories, such as version control systems and bug trackers, foster collaboration and provide means to control and reflect on the development processes.

With the increase in team size and tool support, the amount of data that breaks in on individual developers has grown to a point where it becomes harder and harder for them to grasp implicit relationships among artifacts stored in different locations. Too much time is lost in distinguishing useful information from random noise. In consequence, software engineers are hardly able to fully exploit all their tooling and productivity gains are thus wasted. A new generation of tools is therefore needed—tools that can make use of the semantics of the underlying data to automate tedious processes and filter irrelevant information. The Semantic Web provides a framework to build such tools.

In this paper, we have investigated the research question how software evolution knowledge can be adequately represented by means of ontologies. As an answer to this question, we presented SEON, a family of ontologies that describe many different facets of a software's life-cycle. SEON is unique in that it is comprised of multiple abstraction layers. Our ontologies provide a shared taxonomy of important software engineering concepts and already have found multiple applications. Three of them were discussed in this paper, and we argued that each application clearly benefits from the use of Semantic Web technologies. SOFAS, our software analysis services platform, used SEON as a formal description of the input and output of its individual services. Our guided-input natural language approach HAWKSHAW exploited the clear semantics of OWL to translate program comprehension questions formulated by developers in quasi-natural language to the formal Semantic Web query language SPARQL. This was possible, since the natural language annotation layer of SEON bridged the gap between

machine-processable and human-understandable knowledge. SVB, our Semantic Visualization Broker, relied on reasoning and explicit relations to automatically infer suitable visualizations for given sets of data. All of these three applications would have been significantly harder to implement without SEON and the use of Semantic Web technologies.

We only have started to exploit the potential that the Semantic Web could bring for software evolution support. Other researchers have begun to explore the opportunities and we hope that this paper can encourage even more to do so. A next important step is to consolidate other existing ontologies and to come up with layers of abstraction, similar to what we did with SEON. Based on this, software repositories need to be devised that are semantics-aware, *i.e.*, that produce and consume data in the RDF/OWL format, and that expose stable de-referenceable URIs on the Web. When this is achieved, software repositories could ultimately blend into a queryable global information space of interlinked software evolution data.

Acknowledgements

The work presented in this chapter was supported by the Swiss National Science Foundation as part of the ProDoc "Enterprise Computing" (PDFMP2-122969) and the "Systems of Systems Analysis" (200020_132175) projects. The authors would also like to thank Emanuel Giger and the anonymous reviewers of the Computing Journal for their insightful comments.

Talking to your IDE

4

Supporting Developers with Natural Language Queries

Supporting Developers with Natural Language Queries. M. Würsch, G. Ghezzi, G. Reif, H. C. Gall, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), 2010
DOI: 10.1145/1806799.1806827

The feature list of modern IDEs is steadily growing and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, IDEs often still cannot directly answer the questions that arise during program comprehension tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using guided-input natural language resembling plain English. For that, we model data extracted by classical software analysis tools with an OWL ontology and use knowledge processing technologies from the Semantic Web to query it. We use a case study to demonstrate how our framework can be used to answer queries about static source code information for program comprehension purposes.

4.1 Introduction

Program comprehension plays an important role when performing software engineering activities on large bodies of source code. Both industry and research therefore have aimed to integrate various tools into modern integrated development environments (IDEs) to support software engineers in understanding a software system during their daily development and maintenance tasks.

Especially for novice developers, mastering all the powerful features that are delivered by an IDE, such as Eclipse or Visual Studio, requires a great deal of learning effort. When solving a program comprehension task, developers usually have particular questions in mind, such as *"Where is this method called?"* or *"What fields are declared as being of this type?"* [SMDV06, SMV08]. Unfortunately, such questions often can not be answered directly using existing functionality offered by IDEs. Instead, as explained by De Alwis and Murphy in [dAM08], developers first need to map these *conceptual queries* to one or several *concrete queries*. Even if a particular conceptual query is directly supported, novice programmers are often not yet aware of the capabilities of their IDE. For example, although experienced developers can easily answer *"Where is this method called"* with Eclipse, newcomers still need to be aware that the *"Find references..."* feature, hidden in a context menu, is what they are looking for.

Existing approaches to enable the integration of different information sources often do not allow developers to formulate ad-hoc queries. Instead, they need to be explicitly configured to enable new queries. On the other hand, query languages, such as CodeQuest [HVdM06] or JQuery [JV03], allow developers to formulate queries about software engineering artifacts. These languages are usually based on a SQL- or Prolog-like syntax and effectively using them requires learning effort. According to Chowdhury, however, *"the most comfortable way for a user to express an information need is as a natural language statement."* [Cho04]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [Hen94].

In this paper, we present a framework that allows software engineers to use *guided-input natural language* strongly resembling plain English to query for information about a software system. For that, we combine software evolution data provided by EVOLIZER, our platform for software evolution analysis, with Semantic Web technologies for knowledge processing. We focus on queries concerning

static source code information, such as *"How often is this field accessed?"* or *"What are the subclasses of this class?"*, to demonstrate the potential of our approach; but including more data from various software repositories and tools is straight-forward, as we will explain in detail in this paper.

The remainder of this paper is structured as follows: In Section 4.2 we give an introduction to the Semantic Web and discuss the knowledge processing technologies that we use throughout the paper. We present our framework to query static source code information with (quasi) natural language in Section 4.3. Section 4.4 provides a case study evaluation of our approach. Section 5.6 discusses existing work related to our approach and Section 4.6 concludes the paper.

4.2 Background

The technologies originally developed for the Semantic Web have been proven useful whenever knowledge has to be processed by machines. In this paper, we exploit them to bridge the gap between more classical software analysis tools and natural language query interfaces.

Tim Berners-Lee *et al.* [BLHL01] define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* Following this definition, this semantic extension enriches the Web with meta-data describing the semantics of the webpages in a computer-processable way. Before the webpages can be described with meta-data accordingly, an *ontology* has to be defined for the domain of discourse. An ontology formally describes the concepts (classes) found in the domain, the relationships between these concepts and the properties used to describe them [Gru93]. For example, in the domain of an online record shop, we define concepts, such as *Composer*, *Album*, and *Track*; the relationships *composed by* and *has track*, and the properties *has play-time* and *has title*. The meta-data description of a CD is then able to give the data a well-defined meaning, using the concepts, relationships, and properties defined in the ontology. In the software engineering domain, we define concepts, such as *User*, *Developer*, *Bug*, *Module*; relationships, such as *reports bug*, *fixes bug*, and *is assigned to bug*. Since the Semantic Web describes this information based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first

place.

To bring the vision of the Semantic Web into being, the research community came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases. The Resource Description Framework (RDF) [Ke04] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a bug report, a person, a Web page, a CD, a track on a CD, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [Pe08] can be used to query such RDF graphs.

RDF is domain-independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [De04].

OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [PSHe04]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. *Jena*[1] emerged from the *HP Labs Semantic Web Program* and is a Java framework for building applications for the Semantic Web. It provides a programmatic environment for RDF and OWL.

In this paper, we do not contribute directly to the Semantic Web, but we exploit the technologies introduced above to describe and process data. In short, we model software evolution data with an OWL ontology. Then we take, for example, the static source code information that was extracted with our EVOLIZER toolset, and represent it as an RDF graph that is based on this ontology. RDF graphs, in contrast to relations in a relational model, consist of {*subject, predicate, object*}-

[1] `http://jena.sourceforge.net/`

triples which are close to the natural English sentence structure. This enables the transformation from natural English queries into SPARQL queries which can be issued on the RDF graph. In the remainder of the paper, we describe in detail how the RDF graph is generated and how this knowledge representation is exploited to support software developers.

4.3 Approach

Our vision is to provide a convenient and intuitive interface that allows software engineers—and especially newcomers, who are not yet virtuous with their IDE or command line tools, such as grep and awk—to leverage information sources about various aspects of a software system under development. In particular we want to overcome some of the limitations that existing approaches suffer from: We do not want to restrict developers to a set of predefined queries and we do not want them to learn a specific query language for that limited purpose. Instead, we want developers to be able to use a query language that they are already very familiar with: natural language. A natural language interface to an IDE relieves especially novice programmers from the cognitive burden of translating a conceptual query to a concrete one, which might involve navigating through nested or multi-level menus.

In the following, we briefly introduce EVOLIZER, our platform for software evolution analysis. For the sake of this paper, we focus on the infrastructure that is needed to give developers the possibility to query static source code information from within Eclipse. However, we want to emphasize again that our approach can be generalized to many other domains in the software evolution context.

4.3.1 Evolizer

EVOLIZER[2] basically stems from the idea of having a Release History Database (RHDB) [FPG03b] that integrates information originating from various repositories, such as CVS and Bugzilla, in a single database. It facilitates many common preprocessing steps [ZW04] that are necessary when mining such software archives and, in that context, it is comparable with *Kenyon* of Bevan *et al.* [BEJWKG05] or *eROSE* of Zimmermann *et al.* [ZWDZ05].

[2] `http://www.evolizer.org/`

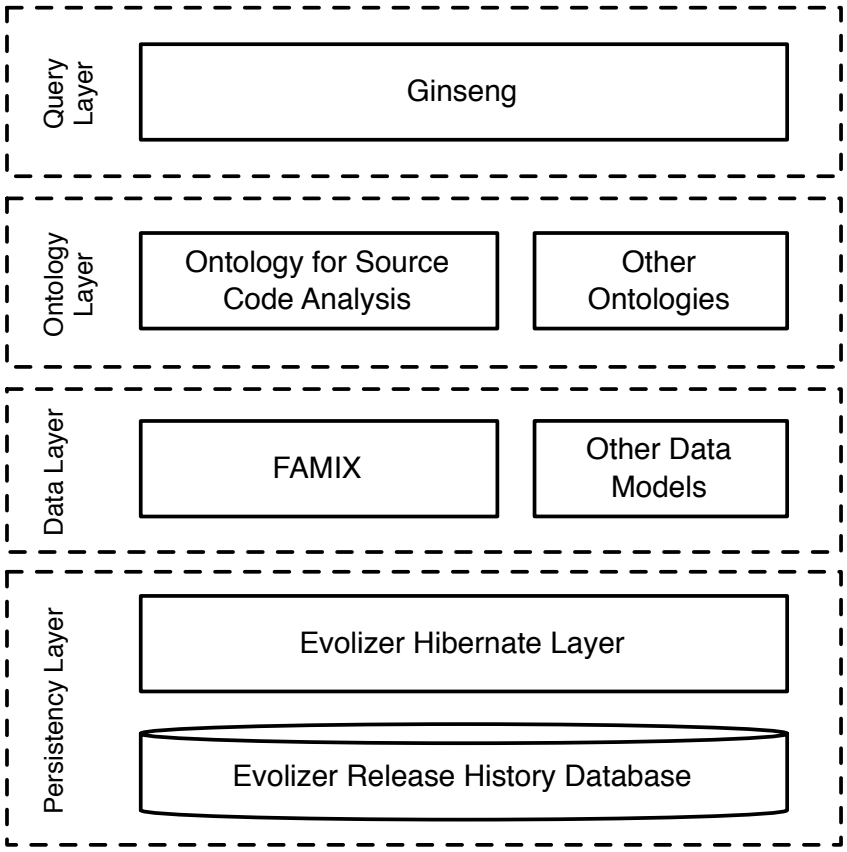


Figure 4.1: The four Layers of Evolizer

Over the years EVOLIZER has advanced from a set of data importers and preprocessors to a platform for various tools that aid developers during their daily maintenance and development tasks. Realized as Eclipse plug-ins, the functionality of EVOLIZER is available at developers' fingertips in a state-of-the-art IDE and incorporates applications that emerged from ideas of the software evolution research community, as well as more classical approaches to support, for example, program understanding. Some of the tools that are built upon EVOLIZER, such as CHANGEDISTILLER [FWPG07, GFP09], follow the original idea of leveraging historical data. Others do not make use of any evolutionary data at all. Instead, for example in case of DA4JAVA [PGKG08], they analyze source code that is currently under development within the IDE.

Figure 4.1 gives an overview on the four layers of EVOLIZER that are relevant for this paper. The persistency layer gives access to facts about a system in a convenient way and provides application support for other EVOLIZER plug-ins to persist settings, query results and so on. In the following, we provide detailed insight into the other three layers: In Section 4.3.2 we describe the *data layer* consisting of a set of data models and data importers. Section 4.3.3 describes the *ontology layer* that enhances the data layer with formally described data semantics. In Section 4.3.4 we show how existing Semantic Web query technology can be used to query an ontology-based knowledge base with quasi-natural language. Section 4.3.5 sums up how the three layers play together to allow a developer to access facts about her source code in a convenient and intuitive way.

4.3.2 Evolizer Data Layer

EVOLIZER provides a set of data models to represent software evolution data along with adequate importer tools to obtain this data from software project repositories. Extending existing data models and data importers, or adding new ones, is straight-forward. For the approach that we present in this paper, we use a tool that was implemented on top of our platform to perform static source code analysis and store the result in our EVOLIZER RHDB.

To add new data, we first identify the key concepts that we want to analyze (in case of analyzing source code: packages, classes, methods, accesses, invocations, etc.) and create a data model accordingly. We use the plug-in extension facilities of Eclipse to make EVOLIZER aware of its data models, so that they can later be accessed by other EVOLIZER plug-ins. For the approach presented in this paper, we plug in a custom-tailored implementation of the FAMIX model [TDD00] to represent facts about source code. Eventually, we need a data importer to extract and store information into the data model that we have registered. In case of our example, this is a parser that extracts the relevant facts from source code under analysis. The FAMIX source code meta-model and the fact extractor that we use are covered in more detail in the next section.

Data models in EVOLIZER are implemented using Java classes annotated with object-relational mapping meta-data according to the Enterprise JavaBeans 3.0

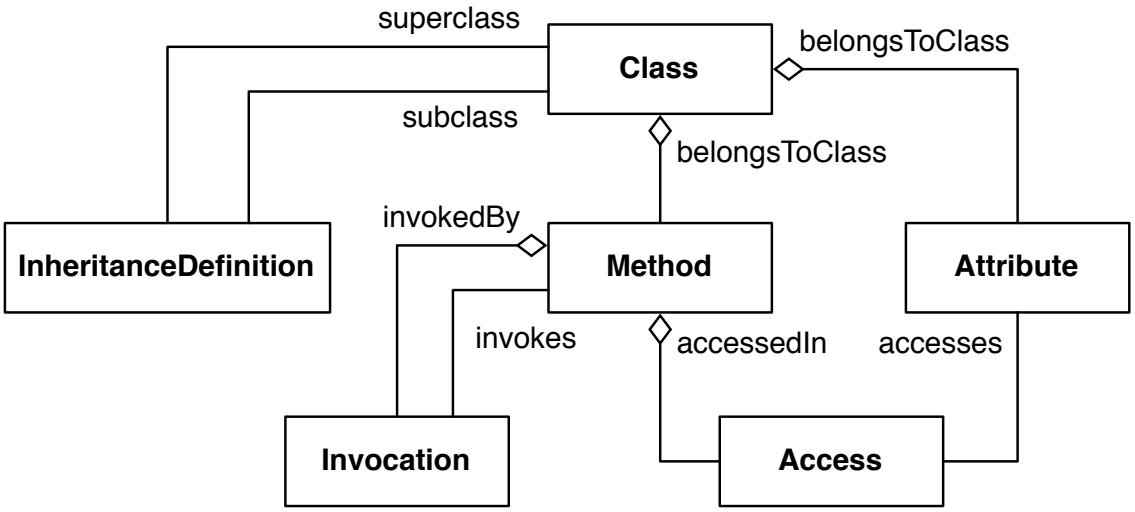


Figure 4.2: Core of the FAMIX Model by Tichelaar *et al.*

(EJB3.0) specification.[3] Persistency is provided through Hibernate,[4] a well-known high performance object/relational persistence and query service. As denoted earlier, EVOLIZER maintains a list of all registered data models and provides means to other EVOLIZER plug-ins to access the evolutionary information of a system via a convenient API.

Evolizer FAMIX

The FAMIX meta-model provides a language-independent representation of object-oriented source code [TDD00] and we use it in EVOLIZER whenever an analysis requires static source code information. An overview of the core model is given in Figure 4.2. It specifies the entities that can be extracted directly from source code.

The model defines important object-oriented relations as classes. For example, *Invocation* is explicitly modeled as a class instead of using a self-aggregation for *Method* (which would be implemented in Java as a collection of *Method*s as an Attribute of the class *Method*). This yields several benefits for us; the most important one is that we can map *Invocations* directly to the EVOLIZER RHDB and query them explicitly later. For example, we can retrieve all *Invocations* that fulfill certain properties, such as that they point to a particular method we are interested

[3] `http://java.sun.com/products/ejb/`
[4] `http://www.hibernate.org/`

```
from Invocation as invocation
    join invocation.callee as callee
where
    callee.name='addChart';
```

Listing 4.1: HQL query to retrieve all Invocations of a Method

in, using HQL—the Hibernate Query Language. An example for such a query statement can be found in Listing 4.1.

The results of such a query can then be used to, *e.g.*, provide a dependency analysis or visualization. EVOLIZER therefore already provides basic SQL-like query access, with features comparable to existing query languages for software analysis, to those that are familiar with both, HQL and the data available in the RHDB. On the other hand, the deficiencies of existing query languages also apply here; for developers that have no deeper knowledge of HQL and the underlying data structures, the information is hardly accessible through the data layer. In Section 4.3.3, we therefore describe how we add another layer to EVOLIZER to enable natural language interfaces.

We rely on a custom implementation of the FAMIX model, realized according to the procedure that we have outlined in Section 4.3.2. To populate an instance of the model with concrete data from source code under analysis, we use ZBINDER by Pinzger *et al.* [PGG07]. ZBINDER builds upon the Java Development Tools (JDT)[5] of Eclipse. The JDT parser alone fails in resolving cross references, such as method calls and attribute accesses of statements that contain a compile error—which is often the case when the code is incomplete or referenced libraries are missing. ZBINDER overcomes this limitation in most instances by gathering additional information stored in the abstract syntax tree and using sophisticated heuristics to reconstruct unresolved method calls.

Static source code information for a software system can either be extracted from past releases that are stored within the EVOLIZER RHDB, or directly from a project that is currently under development in the workspace of Eclipse. ZBINDER can even be registered as a *builder* for a project, so that it gets notified every time a change is made to the source code.

The data layer of EVOLIZER provides a strong foundation for most of the clas-

[5] http://www.eclipse.org/jdt/

Class: Class	**Class: Method**
→ hasMethod Method	→ accessesAttribute Attribute
→ hasAttribute Attribute	→ hasParameter Parameter
→ isReturnTypeOf Method	→ invokesMethod Method
→ isSubclassOf Class	→ hasReturnType Class
→ isSuperclassOf Class	→ isInvokedByMethod Method
→ hasName String	→ isMethodOf Class
	→ hasName String
Class: Attribute	
→ isAttributeOf Class	**Class: Parameter**
→ isAccessedByMethod Method	→ isParameterOf Method
→ hasName String	→ hasName String

Table 4.1: Simplified Version of the Evolizer Ontology for Source Code Analysis

sical software evolution and engineering analyses, especially when they depend on database performance, *e.g.*, interactive software visualizations. Knowledge processing tasks and tool integration, on the other hand, would benefit from formally defined data semantics that the data layer alone can not provide.

In the next section, we demonstrate by example how we overcome this limitation by adding an ontology layer to our platform.

4.3.3 Evolizer Ontology Layer

The EVOLIZER Java implementation of the FAMIX meta-model does not explicitly describe the formal semantics that is needed for automatic knowledge processing tasks such as query answering. For example, we can not define that there is an inverse relation between the property *declares Method* of a *Class* and the property *is declared in Class* of *Method*. If we are able to explicitly state the formal semantics then, every time we make a statement, such as *A declares B*, a semantic reasoner would be able to automatically infer that also *B is declared in A*. The Web Ontology Language OWL allows us to use description logic expressions to describe such relationships and to reason about them.

To take advantage of Semantic Web technologies, we added an additional layer on top of the EVOLIZER data layer by defining an OWL ontology that represents the FAMIX meta-model in terms of OWL classes, relationships and properties. This *source code ontology* is a subset of our software evolution ontology called

```
@rdf("http://evolizer.org/ont/java#Class")
public class FAMIXClass {

  @rdf("http://evolizer.org/ont/java#hasName")
  public String getName() {
    // ...
  }

  @rdf("http://evolizer.org/ont/java#hasMethod")
  public Set<FAMIXMethod> getMethods() {
    // ...
  }

  /* attributes, setter methods, and
     additional behaviour */
}
```

Listing 4.2: Java Class annotated with @rdf.

SEON.[6] Instance data is represented by RDF graphs. This way we get a knowledge base whose formal semantics enables automated processing. An overview of the ontology is shown in Table 4.1. The full ontology covers many more concepts, such as *interfaces*, *local variables*, *type casts* and usages of the *instanceof*-operator, as well as *exceptions*, but for the sake of this paper, we only focus on key concepts.

To populate the knowledge base, we need to map our Java implementation of the FAMIX meta-model to the OWL ontology. This mapping is done via a custom Java annotation `@rdf`. We add an annotation with the URI of the according OWL class to the signature of each Java class that has a counterpart in the ontology. Similarly, we annotate each Java method that should be mapped to a corresponding OWL relation or property name. We use Java reflection to automatically generate RDF statements from Java instances. This approach is similar to—and partially inspired by—the *so(m)mer*-project,[7] an Object-to-RDF mapping layer that uses annotations in the spirit of Hibernate. In Listing 4.2 we show an example of an annotated Java class. We have omitted the EJB3.0 annotations for persistency that

[6] SEON is available for download at `http://www.evolizer.org/`
[7] `https://sommer.dev.java.net/`

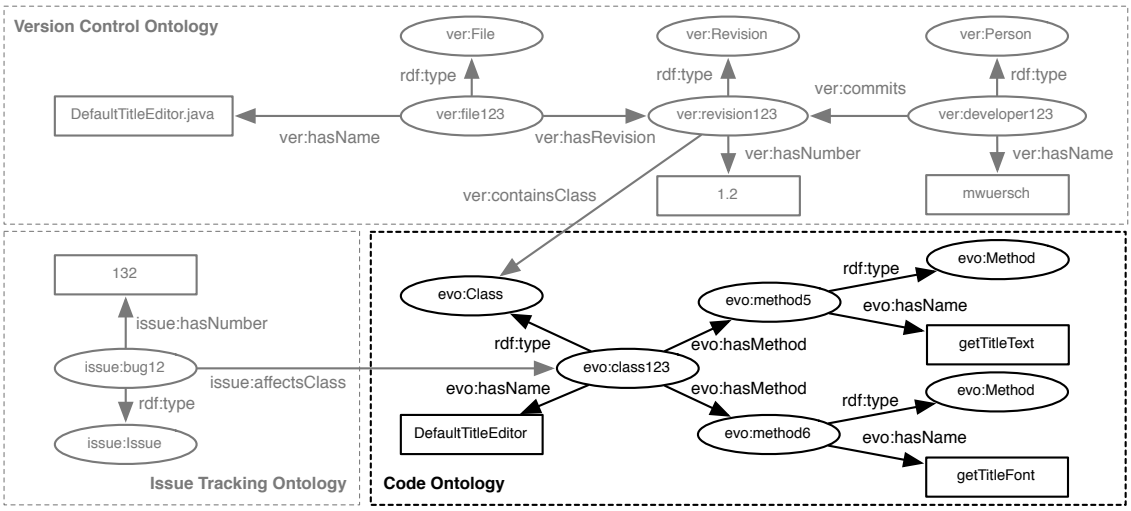


Figure 4.3: Excerpt of a generated RDF Graph.

are used by the data layer of EVOLIZER.

For the following discussion, we introduce the namespace prefix `evo:` as shortcut for `http://evolizer.org/ont/java#`. In the example, the Java class `FAMIXClass` is annotated with the URI `evo:Class` and therefore, for every instance of the `FAMIXClass`, an instance of the OWL class `evo:Class` is generated in the RDF graph. This is done by creating a resource in the graph and adding a `rdf:type` property with `evo:Class` as value. The URI that represents the resource is generated using the unique database id maintained by the data layer of EVOLIZER.

In addition, for each annotated method in the Java class, a property is added to the resource. The return value of the annotated method is used as value of the property in the graph. Since the `getName()`-method has the return type `String`, the value is added as a literal. For the `getMethods()`-method the return type is a set of instances of `FAMIXMethod`. Since the return type is a `Set`, we add a property for each element in the set. The elements in the set are instances of the `FAMIXMethod` class (which is also annotated with `@rdf` annotations). Therefore, we trigger the generation of the RDF statements for the `FAMIXMethod` class as well, repeating the process described above.

An excerpt of the generated RDF graph is shown in Figure 4.3. Ellipses represent resources in the graph. The labels in the ellipses are the URIs that uniquely identify the resources. The labeled arcs represent the properties that relate the

```
@rdf(
   isPredicate=true,
   value="http://evolizer.org/ont/invokesMethod"
)
public class Invocation {

    @subject
    public FAMIXMethod getCaller() {
       // ...
    }

    @object
    public FAMIXMethod getCallee() {
       // ...
    }

    /* attributes, setter methods, and
       additional behaviour */
}
```

Listing 4.3: Java Class that models a Property.

subject and the object of a RDF statement. Finally, literals are depicted as rectangles. In addition to the example that we have outlined above, we list also additional subgraphs that belong to the *version control* and *issue tracking* ontologies in Figure 4.3. This gives an impression on how we integrate different kinds of data sources, although we focus solely on knowledge covered by our code ontology in this paper.

In our Java-to-OWL mapping, we have to take a special case into account. Not all Java classes have counterparts among OWL classes. Java classes that explicitly model relationships are usually modeled as properties or relations in the ontology. In the RDF graph, they are represented as predicates. For example, the *Invocation*-class in the FAMIX model is modeled as *invokesMethod*-property in our ontology. We overcome this clash of paradigms by making it possible to mark a Java class explicitly to model a property by setting the flag `isPredicate` to `true`. In addition, we introduce two additional Java annotations `@subject` and `@object` to specify that, for example, in case of `Invocation`, the method `getCaller()`

returns the subject and `getCallee()` the object of the RDF statement. The value of the `@rdf`-annotation is then considered to be the URI of an OWL-property, rather than of an OWL-class. This renders our mapping approach much more flexible, especially when the underlying Java class models were influenced by a relational view. An example of a class that models a property is given in Listing 4.3.

4.3.4 Evolizer Query Layer: Natural Language Querying with Ginseng

So far, we have discussed how we import static source code information (and facts about the evolution of a system in general) into our EVOLIZER RHDB. We also showed how we use ontology information to augment this data. We explained that we rely on the established industry standard OWL/RDF which enables us to leverage the potential of EVOLIZER with a plentitude of existing tools and frameworks for knowledge processing from academia, as well as from industry.

One such tool is Ginseng, a **g**uided **i**nput **n**atural language **s**earch **eng**ine, that was presented by Bernstein *et al.* in [BKKK06]. Ginseng benefits from the fact that ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles the way how humans talk about things. It can be exploited to use quasi-natural language queries for accessing any OWL/RDF knowledge base. Ginseng is lightweight compared to classical full natural language interfaces since it uses no predefined vocabulary and queries are not interpreted logically or syntactically. Instead, the vocabulary is derived from the OWL knowledge base itself, *i.e.*, from labels of the instance data and from the identifiers that were used to define the ontologies. Optionally it is possible to add synonyms by annotating the ontology with Ginseng tags.

Ginseng uses a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded. The static part additionally contains information on how to translate query sentences from quasi natural language to SPARQL. To generate the dynamic part of the grammar, the ontology is loaded into a Jena inferencing model and for each OWL class, individual (instance), object property, and data type property, a grammar rule is generated.

The full grammar is then used by Ginseng to guide its users by offering an

```
SELECT
      ?methods
WHERE {
      ?methods <rdf:type>       <evo:Method> .
      ?class   <evo:hasMethod> ?methods .
      ?class   <rdf:type>       <evo:Class> .
      ?class   <evo:hasName>    "DefaultTitleEditor" . }
```

Listing 4.4: SPARQL Query that returns the Methods of DefaultTitleEditor

auto-completion feature, *i.e.*, it presents a popup box with suggestions on how to complete the word that the developer is currently typing into the free-form input field. This limits the questions a developer can ask to a certain extent but prevents her from entering queries that are grammatically incorrect. Once the complete query is entered and concluded by a question mark, it is translated by Ginseng into SPARQL statements and executed against the knowledge base maintained by Jena. The results of the query are then presented to the developer.

Consider again the example graph given in Figure 4.3. If we want to query for all the Methods that are declared in the class `DefaultTitleEditor`, we could ask:

What are the methods of DefaultTitleEditor?

That question does not need to be reformulated to a more formal query—it is accepted by Ginseng as it is listed above and developers additionally receive guidance in query composition when they start to type. By *guidance*, we mean that, in case of our example, as soon as the letter *'W'* is typed, all the words starting with that letter are listed in a drop-down menu (see Section 4.4.1 for a full, illustrated example). Ginseng continues to do so until a complete and valid sentence (in terms of that it satisfies the grammar rules) was entered and will then automatically continue with translating the question into the SPARQL query given in Listing 4.4.

When the query above is executed, the graph pattern consisting of the four triple patterns in the `WHERE`-clause is matched against the triples in the RDF graph, and returns the bindings for the variables in the `SELECT`-clause. In SPARQL, variables are indicated by the prefix "?". The predicate and the object of the first triple are fixed values, so the pattern is going to match only triples with those

values. Within a graph pattern a variable must have the same value no matter where it is used, so the query above only returns a binding for `?method` if the variables called `?method` in the first and second triple have the same value; as well as the variables called `?class` in the second, third, and fourth triple.

For more details on Ginseng, we refer to [BKKK06].

4.3.5 Wrapping up: The Integration of the three Layers of Evolizer

When we want to query for facts about source code, we tell the data layer of EVOLIZER to parse the currently selected project in the Eclipse workspace (alternatively it is possible to process the code of a past release stored within the RHDB). The most convenient way to trigger this process is to add a EVOLIZER *Nature* to an Eclipse project. Along with the *nature*, a builder is then assigned to the project that automatically re-builds the FAMIX-model every time a change to the source code is made. Re-building the FAMIX model means that the source code is parsed by `ZBinder` which creates instances of `FamixClass`, `FamixMethod`, `Invocations`, and so on, according to the facts that it finds within the source code. Then it stores these instances into the RHDB.

The data is then passed to the ontology layer which translates it according to the `@rdf`-annotations and the OWL description of the *Evolizer Ontology for Source Code Analysis* into a Jena Ontology model.

This model is then analyzed by Ginseng and, subsequently, available to the developer for querying in natural language. The results are presented to the developer in an Eclipse view, similar to that provided by Eclipse itself for displaying Java search results. Since we also keep track of source code locations in our model, the developer can easily navigate from the results view directly to the corresponding source code.

Next, we provide a case study to demonstrate how our prototype implementation of the framework described above can be used by a developer to answer common questions during daily program comprehension tasks.

4.4 Case Study

In our case study, we demonstrate by the example of the open-source library JFreeChart[8] that our framework can be used to answer the most common program comprehension questions that arise during software evolution tasks [dAM08]. We do not focus on evaluating the quality of the query results—as we have explained throughout the preceding sections, the data importers are not the key contribution of this paper and can be exchanged easily thanks to EVOLIZER's plug-in architecture. Instead, we show that, compared to existing tools, developers are given more flexibility when composing queries with our approach: they can formulate queries conveniently using different variations of natural language sentences.

In [KB07], Kaufmann *et al.* presented a usability study with 48 users, evenly distributed over a wide range of backgrounds and professions, including software development. The study incorporated four query interfaces (including Ginseng) featuring four different query languages that demonstrated the usefulness of natural language interfaces for casual end-users. Their experiment was based on geographical data encoded in an OWL knowledge base. Kaufmann *et al.* found that:

> *"(1) With full-sentence questions, users can communicate their information need in a familiar and natural way without having to think of appropriate keywords in order to find what they are looking for. (2) People can express more semantics when they use full sentences and not just keywords." [KB07]*

Although we did not yet conduct an extensive user study in the software engineering domain, we claim that the results of this study are, to some extent, applicable to our setting. It is reasonable to assume that the domain of the knowledge that we query can be neglected, compared to the influence of the professional background of the users and, as a consequence, their familiarity with more formal languages. The study of Kaufmann *et al.*, however, showed that the findings above apply to both categories of users likewise—to those without any prior knowledge of query languages, as well as to those with a background in software engineering and familiar with at least SQL.

[8] `http://www.jfree.org/jfreechart/`

4.4.1 Using Evolizer to answer common Program Comprehension Questions

In [dAM08], De Alwis and Murphy have listed 36 common program comprehension questions that their tool *Ferret* implements. The questions fall into five categories: *inter-class*, *intra-class*, *inheritance*, *declarations*, and *evolution*. The questions are further assigned to one or several contexts, or what they call contributing *spheres*: The *static*-sphere relies on static program analysis, the *dynamic*-sphere uses profiling information, the *evolution*-sphere relies on software repository mining, and the *plug-in*-sphere contains declarative information specified in Eclipse plug-in manifests.

EVOLIZER supports all of their static queries *out of the box*, without having them predefined or hard-coded explicitly. Conceptually, we can also answer all the questions from the *evolution*-sphere.

In the following we have selected, for each of the first four categories, those questions that proved to be most useful to developers in the field-study conducted by De Alwis and Murphy. We use them to exemplify how EVOLIZER can be used to support program comprehension. As a case study, we use Release 1.0.12 of JFreeChart, an open-source chart library written in Java with a size of more than 250 kLOC.

Questions concerning Inter-Class Dependencies

De Alwis and Murphy identified the question *"What calls this method?"* to be the most commonly asked one when a developer is trying to understand a system. The question falls into the category of inter-class dependencies and can be easily answered with EVOLIZER, as well as by many existing tools—including Eclipse itself. We have randomly chosen the class `ChartDeleter` from JFreeChart. The class declares four methods, among them `addChart(String)`. To find its callers, we can enter exactly:

What methods call addChart?

into the input field of Ginseng and execute the query. Figure 4.4 illustrates how Ginseng provides guidance for the developer to compose a query: When she starts to type *"W"*, a list of possible question words pops up (Step 1). After selecting

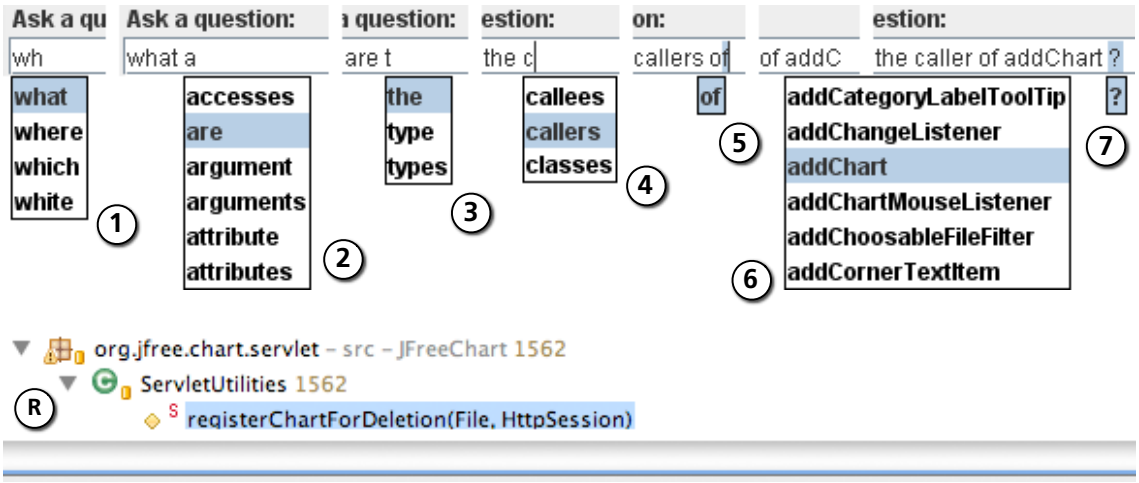


Figure 4.4: Entering a Query and retrieving the Result.

the word *"What"*, she types *"a"* and receives several suggestions starting with that letter, such as *"accesses"*, *"are"*, *"arguments"*, and so on. Going on like that, the developer is able to compose the complete query (Steps 3 to 6) and, as soon a valid query was entered, she can execute it by concluding the sentence with a question mark (Step 7). This kind of guidance is especially valuable for novice programmers, who are not already familiar with the underlying knowledge base.

In case of JFreeChart, a single match is presented after the execution of the query: `registerChartForDeletion(File, HttpSession)` of the Java class `ServletUtilities` (Step R). This corresponds to the result of invoking the *"Find references in project"*-functionality of Eclipse.

Variations of the initial question are also possible:

> *What are the callers of addChart?*

is accepted by Ginseng just as well as the imperative form:

> *Give me the methods invoking addChart!*

We want to remark that Ginseng automatically generates these variations solely based on its grammar rules, synonyms encoded in the ontology, and instance data provided by EVOLIZER. There is no need to explicitly define in advance the questions that are possible—developers can ask them based on the facts extracted from source code.

Another program comprehension question that was identified to be asked often by developers is *"What fields are declared as being of this type?"* The structure of the original question is too complex for the simple grammar rules of Ginseng but can be reformulated to: *"What fields have the type <type>?"* For example, we can search for attributes of type `JTextField` to identify classes of JFreeChart that contribute to the user interface in general and, in particular, process user input. Entering the question:

What attributes have the type JTextField?

returns seven results; three in `DefaultAxisEditor`, two each in `DefaultNumberAxisEditor` and `DefaultTitleEditor`.

This time significantly more user interaction is necessary to come up with the same results using the *Java Search* of Eclipse. Besides entering the search string, we have to choose *Search for type* and configure the *"Limit To"* option to only match for *field types*. Especially newcomers to Eclipse are often not aware of these features.

Questions concerning Intra-Class Dependencies

During maintenance tasks, developers often face god-classes several thousand lines of code in size. Changing, *e.g.*, the type of one of their attributes is a tedious task, involving careful code investigation and a lot of scrolling to answer questions, such as *"What methods access this field?"* Often, the task is further complicated when information hiding principles were violated. Using our framework, we are able to significantly narrow down the amount of code that has to be inspected. Coming back to one of the examples in the last section, we can ask what methods access the field `labelFontField` in `DefaultAxisEditor`. Again, the user is guided during query composition. When she starts typing *"What method accesses..."*, Ginseng automatically suggests *attribute* (as well as field, as a synonym) and *method* as the next possible words. By choosing *attribute*, the set of suggested candidates for the concluding word is reduced to the names of particular fields, *i.e.*, names of methods are faded out. The full query is:

What method accesses the attribute labelFontField?

Executing the query yields two results: the constructor of `DefaultAxisEditor` and the method `attemptLabelFontSelection()`. Manual inspection confirms these results to be correct.

Questions concerning Inheritance

Generalization and specialization are among the most powerful features in Object-Orientation. The other side of the coin is that inheritance increases the gap between the static program and the dynamic process and therefore complicates program understanding in some cases, especially if used too extensive (*e.g.*, deep inheritance hierarchies over more than seven levels), or incorrectly (*e.g.*, to simply reuse code, rather than backed by the idea of specialization). Supporting the developer with tools to understand inheritance hierarchies more easily is therefore desirable. Browsing through code and navigating upwards the inheritance hierarchy is already well-supported by modern IDEs. Navigating downwards, on the other side, usually involves tedious searching. However, literally speaking, it could be as easy as asking *"What are the subclasses of this class?"*, if our approach were used.

The class `DefaultAxisEditor` in JFreeChart is implemented as a subclass of `JPanel`. Querying the static source code information in FAMIX, we can quickly locate similar classes, *i.e.*, classes that extend `JPanel`: `DefaultTitleEditor`, `DefaultAxisEditor`, and `FontChooserPanel`, among others. If the underlying ontology provides meaningful synonyms, each of the following queries would return the information that we are interested in:

- *What are the classes that extend JPanel?*
- *What are the subclasses of JPanel?*
- *What classes inherit from JPanel?*

Questions concerning Declarations

We have chosen *"What are all the fields declared by this type?"* among the 36 conceptional queries that De Alwis and Murphy have listed in their paper to conclude our case study on how developers can benefit from our framework. The question has been identified by De Alwis and Murphy to be the most commonly asked one concerning declarations. In the last example, we have identified a few classes that are subclasses of `JPanel`. If we want to confirm the initial impression that `DefaultTitleEditor` and `DefaultAxisEditor` implement similar concepts according to their naming scheme, we can do that in terms of comparing their states, *i.e.*, fields. Investigating the answers to the queries:

What attributes are defined in DefaultAxisEditor?

and:

What are the attributes of DefaultTitleEditor?

confirms that they, in fact, have similar states: For example, both of them seem to have associated a font (`labelFont` and `titleFont`, respectively), a checkbox (`showTickLabelsCheckBox` and `showTitleCheckBox`, respectively), and so on. The next steps would probably be to investigate further the types of the fields and the behavior that operates on them or to have a look at the documentation of the two classes, *e.g.*, by entering the following query:

Give me the Javadoc of DefaultAxisEditor!

From here, gaining a deeper understanding of JFreeChart is just a matter of asking the right questions.

4.4.2 Discussion and Limitations

We conducted a validation of our approach by addressing the questions that De Alwis and Murphy listed in their paper about Ferret. The two approaches share many similarities in terms of their goals. Furthermore, those questions were identified in two empirical studies [SMV08, SMDV06] to be the most frequently asked questions by programmers during software evolution tasks and therefore provide a suitable benchmark for our framework.

Our approach, in contrast to Ferret, can draw from the full power of the semantic web technologies and is therefore not limited to a set of predefined queries, with the need to have any additional ones implemented by some provider, such as a tools-support group. Instead, the querying capabilities of our framework are much more flexible and only limited by a subset of the English grammar and by the knowledge base that is available. Therefore the developer queries that we have chosen in our case study are only a subset of the ones that can be formulated and answered *out of the box*. Many more are possible and they can be formulated in different variations, *e.g.*, as a questions or using the imperative form.

IDE vendors need to provide an interface to the information offered by tools comparable to Ferret (often menu-items in deeply nested context-menus). This

```
from
      Method caller, Method callee
where
      caller.calls(callee)
and   caller.fromSource()
and   callee.fromSource()
and   caller != callee
and   callee.getName().matches("addChart%")
select
      caller.getName()
```

Listing 4.5: Query to retrieve the Callers of a Method with Semmle

becomes more and more of a problem as the information need of developers may become more diverse with the increase in complexity of modern software systems. Our framework, in contrast, provides a single access point for most of the information needs: using natural language, a developer can just ask what is on her mind, without having to worry where the desired functionality is hidden in the deeply nested menus of her favorite IDE.

Existing query languages for software evolution artifacts potentially also provide such an access point and give the developer the freedom to formulate queries without being bound to a set of predefined ones. On the other hand, they usually rely on custom-tailored, verbose languages. Therefore, they are hardly used in practice. A simple question, such as *"What methods call addChart?"*, which our tool answers right away, has to be reformulated by a developer into a SQL-like statement in order to be answerable with a tool, such as Semmle.[9] A Semmle query would look like the one in Listing 4.5.

Moreover, extending existing query languages with new vocabulary involves manual editing of the language definition files, whereas in our framework, additional vocabulary is available as soon as new data is loaded into EVOLIZER. This is possible because Ginseng generates dynamic grammar rules from the loaded OWL ontologies, but it also implies that we have to rely on meaningful identifiers in the ontologies that we query. If this is not the case, we also have to fall back to manual definitions of synonyms. This is straight-forward and can be done either

[9] http://www.semmle.com/

in advance by a tool-vendor or later by the end-users, *i.e.*, developers—even if they are unfamiliar with the Semantic Web—in case that they are more comfortable with another vocabulary than the one that is already provided by the ontology.

Query languages are less ambiguous than natural language in general and therefore better in expressing, *e.g.*, complex restrictions and in formulating composed queries. However, supported by the empirical studies mentioned above, we claim that the most common program comprehension questions have a simple structure. In rare cases where more expressive power is needed, one can always fall back to SPARQL or to the SQL-like Hibernate Query Language (HQL).

The performance of our prototype on a common laptop computer is acceptable for a project of the size of JFreeChart (∼250kLOC, the response time for the queries presented in our case study was usually around a couple of seconds).

4.5 Related Work

Our work is highly related to the approach of De Alwis and Murphy presented in [dAM08]. Just like them, we offer a framework to support the composition and integration of different sources of data about software artifacts in a single queryable knowledge base. In contrast to our approach, they define their own *sphere model*, whereas we rely on standardized technologies that are already established in the research community, as well as in industry. Moreover, while Ferret restricts developers to a set of predefined, hard-coded questions, we give them the freedom to formulate their own questions by exploiting existing natural language query tools.

4.5.1 Natural Language in Program Comprehension

LaSSIE, presented by Devanbu *et al.* in [DBS91], integrated multiple views on a software system at AT&T in a frame-based knowledge base and also provided semantic retrieval through a natural language interface. LaSSIE and our framework share many commonalities, especially since the Semantic Web emerged from frame-based knowledge representation techniques. Hill *et al.* presented an algorithm to extract noun, verb, and prepositional phrases from method and field signatures in source code to enable *contextual searching* [HPVS09]. The queries they support are closer to keyword search on identifiers found in source code than to

full natural language questions and they do not cover structural information, such as caller-callee or inheritance relationships among source code entities.

4.5.2 Query Languages for Software Artifacts

Many approaches have been proposed that use specific languages to query software artifacts. They are either based on standard database languages, such as SQL or Datalog (*e.g.*, CodeQuest [HVdM06] and Semmle), customized Prolog implementations (*e.g.*, JQuery [JV03], ASTLog [Cre97], GraphLog [CMR92]), or a custom language (*e.g.*, SCA [PP96]). All of them aim to help developers to effectively explore and better understand code, uncovering information that would be impossible or extremely hard to find with standard tools. However, most of them require the user to master syntax and vocabulary of a specific query language limited to that single purpose. Our approach guides developers in vocabulary, as well as in syntax, to construct well-formed and coherent questions about static source code information. Nevertheless, we consider most of these query languages complementary to our approach, as they are more expressive in terms of that it is possible to compose more complex queries than with the subset of English grammar rules that we rely on. In general, as argued in Section 4.4, the most common questions that arise during software evolution tasks are of simple structure and are therefore predestinated to be answered with natural language using EVOLIZER.

4.5.3 Semantic Web in Software Engineering

Our framework relies heavily on Semantic Web technologies. Besides the Web, these technologies have proven to be useful in many domains, for example to enable the interoperability of software systems, and when technologies are needed to express knowledge with formal semantics to enable machine processing. Software Engineering is one of these domains. An overview of applications of ontologies in software engineering has been given in [HS06, GL02, UJ96]. All of these publications promote the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. On the other hand, only few approaches put those ideas into real practice. Hyland-Wood *et al.* [HWCK06] propose an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics, and re-

quirements. Bertoa *et al.* [BVG06] follow a similar approach but focus more on software measurement. Happel *et al.* [HKST06] propose various ontologies to foster software reuse. In their KOntoR approach, they provide therefore background knowledge about software artifacts, such as the programming language and licensing models. Kiefer *et al.* developed EvoOnt, a software repository data exchange format based on OWL [KBT07]. Their software ontology model (SOM) was influenced by FAMIX. Their version ontology model (VOM) and their bug ontology model (BOM) are based on EVOLIZER's data models for CVS and bug tracking information, respectively. The authors use iSPARQL, their extension to the RDF query language SPARQL, for effectively querying their ontologies to detect code smells. Witte *et al.* [WZR07] use text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings are represented in RDF. The MOST project [mos] aims to facilitate software engineering by leveraging ontology and reasoning technologies. It integrates ontologies into model-driven software development (MDSD), resulting in ontology driven software development (ODSD). All of the approaches mentioned above acknowledge the potential of ontologies and the Semantic Web applied to software engineering. They often define custom ontologies that can be integrated in the ontology layer of EVOLIZER.

4.6 Conclusions

As software systems get more complex, efficient tools to support software engineers during their development and maintenance tasks are becoming more important. Modern IDEs already made a great leap forward in providing a variety of features to, for example, facilitate program comprehension. The complexity of the user interface is putting a significant cognitive burden on a developer. Often it is easier to solve a task manually than to master a tool. Although experienced developers usually know exactly what information they are looking for, they often do not know how to get it. They simply do not know how to turn conceptual queries into something their IDE understands.

In this paper, we presented a framework that overcomes this gap and showed an application of Semantic Web technologies that goes beyond merely data exchange for the sake of tool interoperability. We combined industrial-strength technologies with ideas and tools from the Semantic Web to enable developers to

query software engineering artifacts in a way that is familiar to them: using (quasi) natural language strongly resembling plain English. For that, we use the Web Ontology Language OWL to describe static source code information extracted by our EVOLIZER. The resulting ontology then serves as input for the guided-input natural language interface Ginseng. We demonstrated in a case study that our approach makes it possible to answer the most common program comprehension questions identified in the literature.

We do not restrict developers to a set of predefined questions but advance the state-of-the-art in that our approach is only dependent on what data is available as input. With our framework, it is straight-forward to integrate almost any kind of evolutionary information, for example, from version control or issue tracking systems—solely by exploiting existing and well-established standards for resource description. We encourage other researchers to download and try out our EVOLIZER toolset or to incorporate our SEON ontology into their own tools.

5 Change Distilling

Fine-Grained Source Code Change Extraction

Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. B. Fluri, M. Würsch, M. Pinzger, H. C. Gall, IEEE Transactions on Software Engineering (TSE), 33(11):725–743, 2007
DOI: 10.1109/TSE.2007.70731

A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. We present *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. For that, we have improved the existing algorithm of Chawathe *et al.* for extracting changes in hierarchically structured data [CRGMW96]. Our algorithm extracts changes by finding both a match between the nodes of the compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. As a result, we can identify fine-grained change types between program versions according to our *taxonomy of source code changes*. We evaluated our change distilling algorithm with a benchmark we developed that consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. We achieved significant improvements in extracting types of source code changes: our algorithm approximates the minimum edit script by 45% better than the original change extraction approach by Chawathe *et al.* We are able to find all occurring changes and almost reach the minimum conforming edit script, *i.e.*, we reach a mean absolute percentage error of 34%, compared to 79% reached by the original algorithm. The paper describes both our change distilling algorithm and the results of our evaluation.

5.1 Introduction

Since Lehman's Laws of Program Evolution from the 1980's [Leh80] it is well understood that software has to be adapted to changing requirements and environments or it becomes progressively less useful. Change is broadly accepted as a crucial part of a software's life-cycle. As a consequence, in the last years several techniques and tools have been developed to aid software engineers in maintaining and evolving large complex software systems. For instance, Ying *et al.* or Zimmermann *et al.* developed approaches that guide programmers along related changes by telling them "programmers who changed these functions also changed..." [YMNCC04, ZWDZ05]. The Hipikat tool of Čubranić *et al.* used project history information to provide recommendations for a modification task [CMSB05]. Gall *et al.* detected possible maintainability hot-spots by analyzing co-change relationships of modules [GHJ98].

We argue that such techniques and tools are valuable but suffer from the low quality of information available for changes. Typically, such information, in particular for source code, is stored by versioning systems (*e.g.*, CVS or Subversion). They keep track of changes by storing the text lines *added* and/or *deleted* from a particular file. Structural changes in the source code are not considered at all.

More sophisticated approaches are able to narrow down changes to the method level but fail in further qualifying changes such as, for example, the addition of a method invocation in the else branch of an if statement. Furthermore, a classification of changes according to their impact on other source code entities is missing. In particular, the latter information is important to improve the quality of software evolution results and as a consequence to provide better support for programmers and system analysts.

Since source code can be represented as abstract syntax trees (AST), tree differencing can be used to extract detailed change information. This approach is promising, because exact information on each entity and statement is available in an AST. In our previous work [FG06] we have built a *taxonomy of source code changes* that defines source code changes according to tree edit operations in the AST and classifies each change type with a *significance level*. The level expresses how strong a change may impact other source code entities and whether a change may be functionality-modifying or functionality-preserving. In our taxonomy, we focus on object-oriented programming languages (OOPLs) and Java in particular.

By adjusting the change type extraction the taxonomy can also be used for other OOPLs. In total, our taxonomy defines 35 changes types.

In this paper, we present *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. For that, we improved the existing algorithm for extracting changes in hierarchically structured data of Chawathe *et al.* [CRGMW96]. This algorithm finds changes according to basic tree edit operations, such as *insert*, *delete*, *move*, or *update* of tree nodes.

The contributions of this paper are twofold: 1) Our change distilling algorithm and 2) a *benchmark* to evaluate source code change extraction algorithms.

Our change distilling algorithm uses the *bi-gram string similarity* to match source code statements (such as method invocations, condition statements, etc.) and the *subtree similarity of Chawathe* et al. to match source code structures (such as if-statements or loops). To further improve the matching, we use a best match algorithm for all leaf nodes and inner node similarity weighting. To overcome mismatch propagation in small subtrees we use dynamic thresholds for subtree similarity.

The second contribution of this paper, the benchmark we developed, consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects to evaluate our change distilling algorithm. Compared to the original change extraction algorithm of Chawathe *et al.*, we perform 45% better. We almost reach the minimum conforming edit script, *i.e.*, we reach a mean absolute percentage error of 34%. With this knowledge about source code changes existing software evolution analysis tools can be improved. For instance, the Hatari tool rates the risk of changing a method according to the frequency of method changes that caused a bug [SZZ05a]. Detailed information about the changes is not taken into account, for instance, whether a bug is caused by the insertion of a method invocation statement or by the insertion of a whole else-if-statement. With the information obtained from CHANGEDISTILLER such a differentiation would be possible: Hatari could inform software developers which change types in which parts of the method body are risky to apply.

The remainder of the paper is organized as follows: In Section 5.2 we present the original algorithm of Chawathe *et al.* and describe inadequacies concerning the extraction of source code changes. Section 5.3 presents string and tree similarity measures and our improved algorithm. We discuss our implementation including the generation of the tree representation in Section 5.4. In Section 5.5

the benchmark and our results are described. Section 5.6 reviews the related work. We conclude the paper with Section 5.7.

5.2 Change Extraction in Tree-like Data Structures

Since source code is represented in a tree-like data structure, *i.e.*, in an abstract syntax tree (AST), we can use tree differencing algorithms to extract changes between two versions of a Java class. We use basic tree edit operations to describe changes applied to source code.

Our algorithm to extract changes is based on the work by Chawathe *et al.* in [CRGMW96]. We discuss the reasons why it is adequate to build up on this algorithm in the related work section (Section 5.6). In the following we introduce the terminology and outline their original algorithm, that outputs an edit script of basic tree edit operations transforming an original into a modified tree. Then, we illustrate why the original algorithm is not adequate for source code and discuss how we improved it to handle source code changes.

5.2.1 Terminology

Speaking in terms of graph theory, a tree is a directed acyclic graph consisting of nodes interconnected by edges representing a parent-child relationship. According to the notation used by Chawathe *et al.*, a node n is the *parent node* of a node m, $n = p(m)$, if m is a child of node n. Nodes along the path to the top of the tree are called *ancestors of* m. In return, m is called their *descendant*. The node in a tree that has no parent is called *root node* or *root*. The nodes that have no children are called *leaf nodes* or *leaves*. Nodes in-between are *inner nodes*. Whenever the distinction between *root*, *inner node*, and *leaf* does not add to our discussion, we will talk about *nodes* in general. A node n has a label, $l(n)$ and a value, $v(n)$. In our graphical tree representation, node labels are put inside a node, *e.g.*, A, and node values left or right beside the node, *e.g.*, *"val."* Figure 5.1 illustrates this terminology with an example tree. We apply a tree differencing algorithm on two ASTs denoted by T_1 and T_2. Leaves in the tree are non-compound statements, *e.g.*, *method invocation*

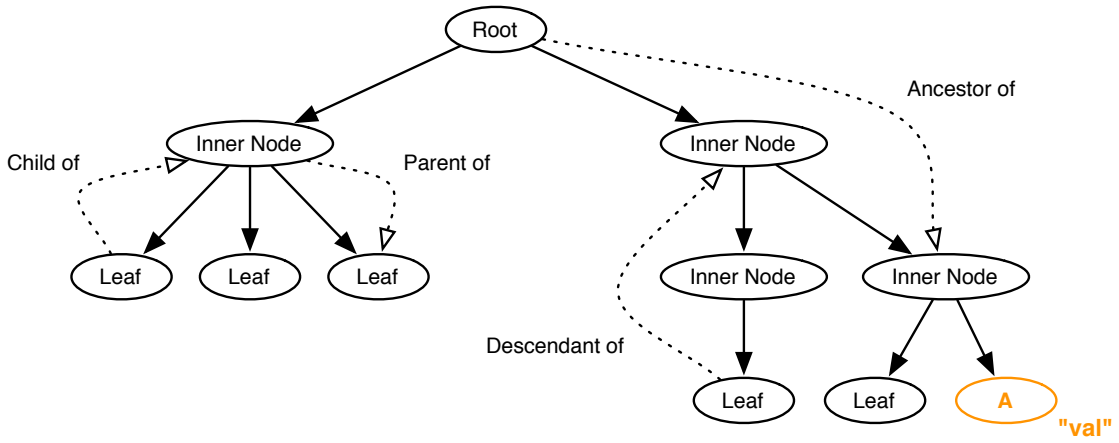


Figure 5.1: A generic Tree Structure. The right-most leaf shows how we annotate labels and values of nodes

or *assignment*. For all nodes, the label is the type of the statement, *e.g.*, *MI* for a method invocation or *IF* for an if-statement. The value of an inner node depends on its label, for instance, the condition expression for if-statements: *"a < b."* For leaves, the value is the textual representation of the statement, *e.g.*, the method invocation statement *"x.foo(arg);"*.

Changes are detected between two trees T_1 and T_2. In general T_1 denotes the original tree and T_2 the modified tree.

5.2.2 Basic Algorithm

Our change detection relies on the algorithm presented in [CRGMW96]. Their algorithm detects changes in hierarchically structured data represented in tree-like data structures. To extract the changes, the algorithm splits the problem into two tasks:

- Finding a "good" matching between the nodes of the trees T_1 and T_2.
- Finding a minimum "conforming" edit script that transforms T_1 into T_2, given the computed matching.

Finding a "good," *i.e.*, correct and accurate, matching between the nodes is *crucial* to the outcome of the edit script task. The more nodes that can be matched, the better the minimum conforming edit script.

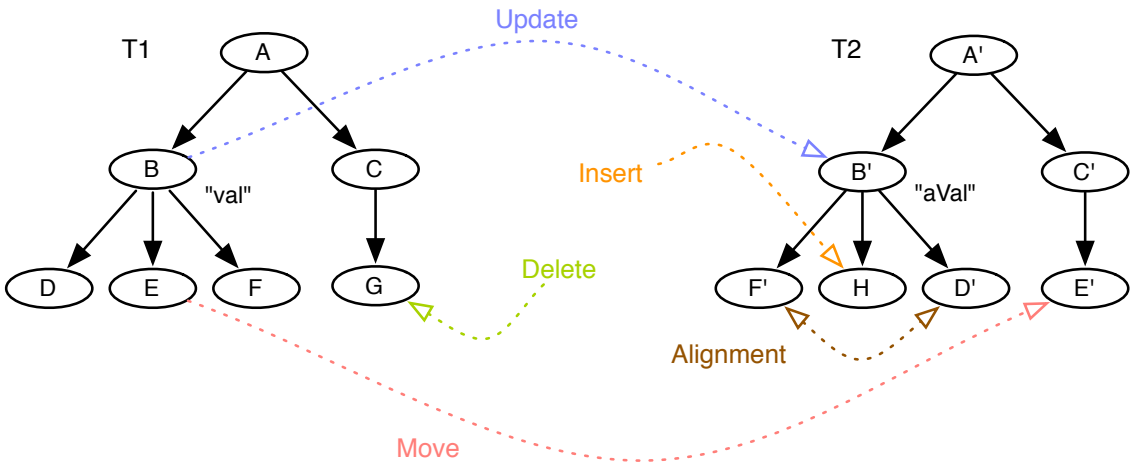


Figure 5.2: The five tree edit operations extracted by the edit script generation algorithm by Chawathe *et al.* Nodes with the same letter are intended to match (example: A matches A'). Node values have been omitted unless they changed from T_1 to T_2.

We first outline the calculation of the edit script and then describe the matching procedure in detail to highlight the parts to be adapted for detecting changes in source code.

Calculating an Edit Script

The matching set of node pairs is passed to the edit script generation which runs through five phases. Each phase is designed to detect one of the following basic tree edit operations, also illustrated in Figure 5.2.

- **Insert**: $\text{INS}((l, v), y, k)$; insert new leaf node with label l and value v as kth child of node y, *e.g.*, in Figure 5.2 H is inserted as child of B': $\text{INS}((H, \text{“”}), B', 2)$.
- **Delete**: $\text{DEL}(x)$; delete node x from its parent $p(x)$, *e.g.*, in Figure 5.2 G is deleted: $\text{DEL}(G)$.
- **Alignment**: $\text{MOV}(x, p(x), k)$; node x becomes the kth child of $p(x)$, *e.g.*, in Figure 5.2 F' becomes the first child of its parent B': $\text{MOV}(D, B', 3)$.
- **Move**: $\text{MOV}(x, y, k), p(x) \neq y$; node x becomes the kth child of y and is deleted from $p(x)$, *e.g.*, in Figure 5.2 E is moved from B to C': $\text{MOV}(E, C', 1)$.
- **Update**: $\text{UPD}(x, val)$; update $v(x)$ with val, *i.e.*, $val = v_{new}(x)$ and $v_{old}(x) \neq v_{new}(x)$, *e.g.*, in Figure 5.2 the value of B is updated: $\text{UPD}(B, \text{“aVal”})$.

Matching Procedure

The matching procedure finds an appropriate matching set of pairs of nodes from T_1 and T_2. Chawathe *et al.* define two fundamental matching criteria necessary for the algorithm to produce a "good" matching set with which a minimum conforming edit script is achieved.

Matching Criterion 1 (Leaves):

$$match_1(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & sim(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

Leaves match if their labels are equal and their values (as strings) are similar according to a given string similarity measure, $sim(x, y)$. The value f is the threshold for the string similarity. Pre-testing the labels for equality is important to prevent the matching of different node types.

Matching Criterion 2 (Inner nodes):

$$match_2(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \text{ and} \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \\ false & \text{otherwise} \end{cases}$$

Where $|x|$ denotes the number of leaves contained by x.

The inner node matching does not use similarities for the node-values. Instead it uses a measure of how many leaves the subtrees have in common:

$$common(x,y) = \{(w,z) \in M | w \text{ is leaf of } x, \text{ and } z \text{ is leaf of } y\}, \text{where } M \text{ is the set of matched node pairs.}$$

The number of common leaves is put into proportion to the maximum number of leaves in either subtrees. The value t is the threshold for the inner node similarity. Matching Criterion 2 puts a strong focus on the leaves and is therefore good for LaTeX documents, where leaves (words, or sentences of natural language) cover most of the text semantics.

Since the approach presented by Chawathe *et al.* is used for detecting changes in hierarchically structured documents, they use an assumption to make a *unique maximal* matching:

> **Assumption 1**: For any leaf $x \in T_1$ there is at most one leaf $y \in T_2$ such that $sim(v(x), v(y))$ 0.

The assumption that there is at most one leaf in the right tree that can match a corresponding leaf in the left tree (and vice versa) is a necessary precondition for the algorithm to produce an optimal matching and a minimal conforming edit script, consequently. Even if the assumption fails, Chawathe *et al.* apply a post-processing step to improve the solution. For source code comparisons, Assumption 1 is one of the main reasons why the approach by Chawathe *et al.* produces suboptimal results. In Section 5.2.3, we discuss the assumption and the post-processing step, as well as the circumstances under which the post-processing step is insufficient for our concerns.

5.2.3 When Matching Fails

When applied to source code, the shortcomings of the basic algorithm impact the matching set—in these cases the matching fails. But *failing* does not mean that the algorithm yields incorrect results, *i.e.*, leading to an edit script that does not transform the original into the modified tree correctly. The edit script is always correct, but if the matching is inadequate, the solution may not minimal.

The quality of the sim-function and the associated threshold f, introduced in the first matching criterion, are crucial for an optimal matching on the leaf-level. When Assumption 1 does not hold, a mismatch on leaves can be propagated to inner nodes, leading to a mismatch on higher levels. This can happen whenever a certain number of children of an inner node violate Assumption 1; this is particularly prominent for small subtrees. In the following, we discuss issues concerning leaf-matching based on node values and illustrate mismatch-propagation.

Node Values

Matching leaves is based on two conditions: the leaves have to be of the same kind, which we can verify by testing their labels for equality. The second condition

```
if (a > b) {
    foo.getHuga();
    foo.doNothing();
}
```
(a)

```
if (a > b) {
    foo.getHuga();
    foo.bar();
}
```
(b)

Figure 5.3: (a) The original if-statement; (b) The modified if-statement: The method invocation `foo.doNothing();` was replaced by `foo.bar();`

applies to the values of the leaves and is evaluated using the function introduced in Matching Criterion 1. In terms of the AST that we use, values correspond to statements (or to the condition in case of an if-statement) which are strings. Consider the two strings *verticalDrawAction* and *drawVerticalAction*, which can be found, *e.g.*, in method invocation statements. From a human's point of view, we intuitively see, that they can be considered as an original and a modified version of the same statement. Especially when they were found in the same context, *i.e.*, in subsequent versions of the same method of a class.

Considering common string similarity measures, context semantics are missing. As we observed in our case studies, common renaming of identifiers during refactoring often involves changing the word order. To allow these strings to match, we have to lower the string similarity threshold f significantly, possibly resulting in false negatives in other places.

Small Subtrees

A mismatch on a single leaf-pair does not have a noteworthy impact on the quality of the outcome of the algorithm; we find additional insert- and delete-operations instead of update-operations in the edit script. But, these mismatches can be propagated to higher levels of the tree, leading to a complete mismatch of a whole subtree and therefore to many unnecessary tree edit operations.

We discuss the propagation of mismatches using small trees as an example: between the code snippets in Figure 5.3 (a) and (b) a single statement was deleted and a new one was inserted. The surrounding code did not change at all and the threshold t of Matching Criterion 2 is set to 0.6. Figure 5.4 visualizes the same source code using an AST representation. The node with label *IF* denotes an if-statement. Its value corresponds to the if-condition. The node with label

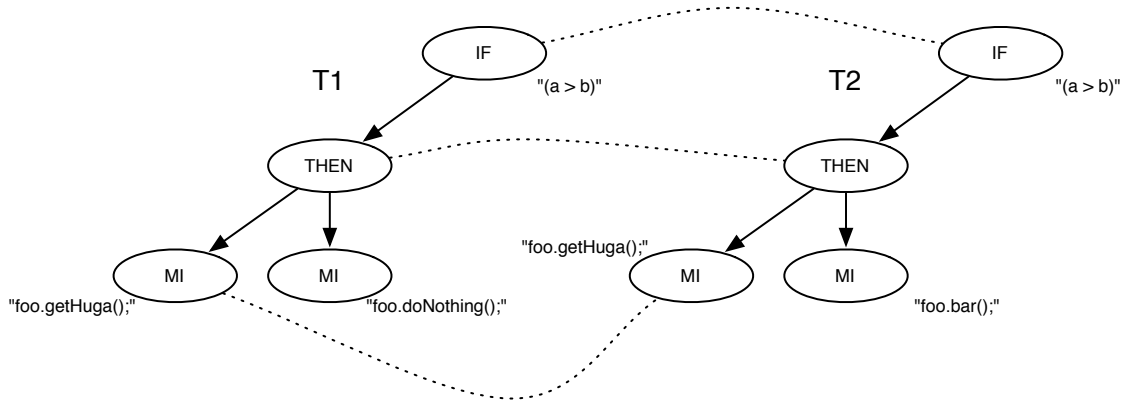


Figure 5.4: An example of two similar trees T_1 and T_2 for which the algorithm fails to calculate a minimal edit script.

THEN denotes the then-block. The node with label *MI* denotes method invocation statements, that are listed as values. For the matching, we traverse the trees bottom-up, *i.e.*, in depth-first manner from left to right. The leaves representing the method invocation `foo.getHuga();` in T_1 and T_2 match according to Matching Criterion 1. They are added to the matching set and marked as *matched*. Although the labels of both right leaves are the same, the values `foo.doNothing();` and `foo.bar();` cannot be matched. We are proceeding to the next level in the tree and reach the inner node representing the then-block in T_1. Inner nodes are matched in accordance to Matching Criterion 2, we count the number of common leaf-descendants of both nodes and divide them by the maximum number of leaves in either trees, leading to the tree similarity of 0.5 and therefore to a mismatch of the two then-blocks: $\frac{|common(x,y)|}{max(|x|,|y|)} = \frac{1}{2} = 0.5$.

We proceed to the root of the subtree, the if-statement, which are not matched due to the inner node similarity of 0.5. The final (mis-)matchings are shown in Figure 5.5. Although the trees in Figure 5.4 show a potential matching set of three node pairs, the algorithm fails—only one node can be matched using the matching criteria and a threshold of 0.6.

When Assumption 1 does not hold

Considering source code, similar statements can occur frequently. For instance, statements that print out a particular string on the console are commonly used

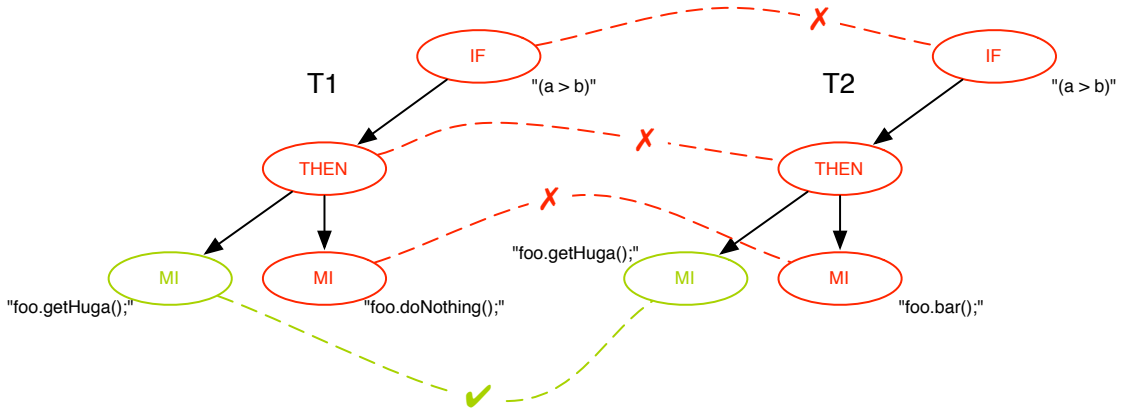


Figure 5.5: The whole subtree is considered as mismatched.

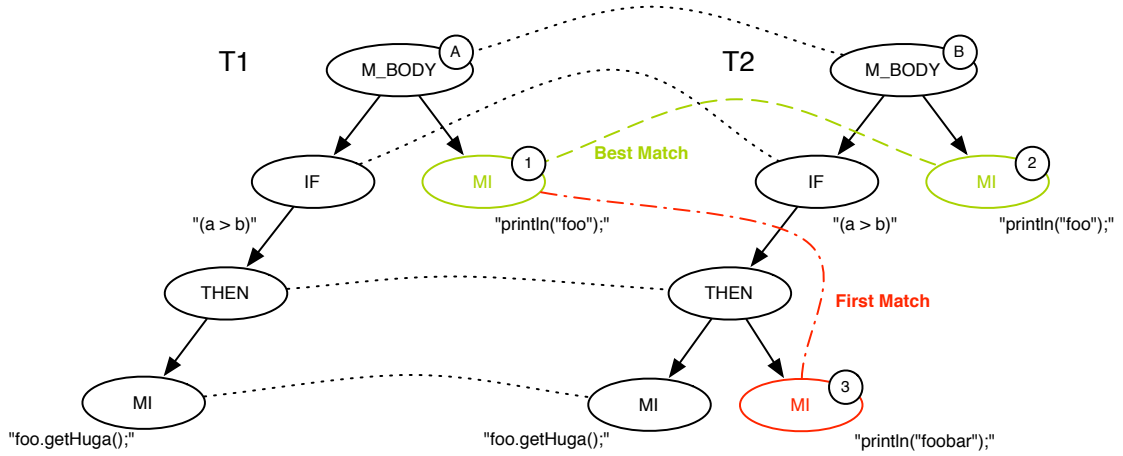


Figure 5.6: Suboptimal results are very likely to occur whenever Assumption 1 does not hold.

for debugging. In such cases, there is more than one matching partner for a single node $x \in T_1$ leading to a violation of Assumption 1. Figure 5.6 shows the consequences that a single statement insert (Node 3) can have: there is more than one possible counterpart in the right tree for Node 1, namely Nodes 2 and 3. Since the tree is traversed in bottom-up manner, Nodes 1 and 3 are put into the matching set, whereas the better match, *i.e.*, the pair of identical Nodes 1 and 2, is not considered to match.

In T_1, the root is the only node that remains. Due to the simplicity of our example, we are able to catch mismatching propagation on this third level: according to Matching Criterion 2, the roots match because they have two common leaves

divided by a maximum of three leaves in T_2, leading to a similarity of $\frac{2}{3}$, which lies above threshold $t = 0.6$. Even for our trivial example, the algorithm found nine changes—eight more than we have expected. We have expected the insert operation INS$((MI, \text{“}println(\text{”}foobar\text{”});\text{”}), THEN, 2)$, but the changes found are:

1. INS$((IF, \text{“}(a > b)\text{”}), M_BODY, 1)$,
2. INS$((MI, \text{“}println(\text{”}foo\text{”});\text{”}), M_BODY, 2)$,
3. INS$((THEN, \text{“”}), IF, 1)$,
4. INS$((MI, \text{“}foo.getHuga();\text{”}), THEN, 1)$,
5. MOV$((MI, \text{“}println(\text{“}foo\text{”});\text{”}), THEN, 2)$,
6. UPD$((MI, \text{“}println(\text{“}foo\text{”});\text{”}), \text{“}println(\text{“}foobar\text{”});\text{”})$,
7. DEL$((MI, \text{“}foo.getHuga();\text{”}))$,
8. DEL$((THEN, \text{“”}))$, and
9. DEL$((IF, \text{“}(a > b)\text{”}))$

In cases where Assumption 1 does not hold, a post-processing step is applied. For each matching pair (x, y), where $x \in T_1$ and $y \in T_2$, it is checked whether the matching partner of a child node c of x is a child node of y. If not, it is checked, whether a child c' of y can be found that $match(c, c')$ holds. In this case the old matching pair is replaced by (c, c'). For further details, we refer to [CRGMW96]. In the example above, the post-processing improves the matching set: for the matching pair $(\text{Node A}, \text{Node B})$, we check whether the matching partner of Node 1 is a child node of Node B. This is not the case. Therefore, we search for an unmatched child c' in Node B so that $match_1(\text{Node } 1, c')$ holds. Node 2 is such a c' in Node B. We replace the matching pair $(\text{Node } 1, \text{Node } 3)$ with $(\text{Node } 1, \text{Node } 2)$. The expected node is matched which reduces the previous edit script by the changes 2, 5, and 6 but adds INS$((MI, \text{“}println(\text{“}foobar\text{”});\text{”}), THEN, 2)$.

There are a number of tree constellations in which the post-processing step does not improve the matching. In Figure 5.7, we show an example of such a constellation. Node 1 has been moved between T_1 and T_2 to a new position: it has been moved two levels up and is represented by Node 2 in T_2. Post-processing

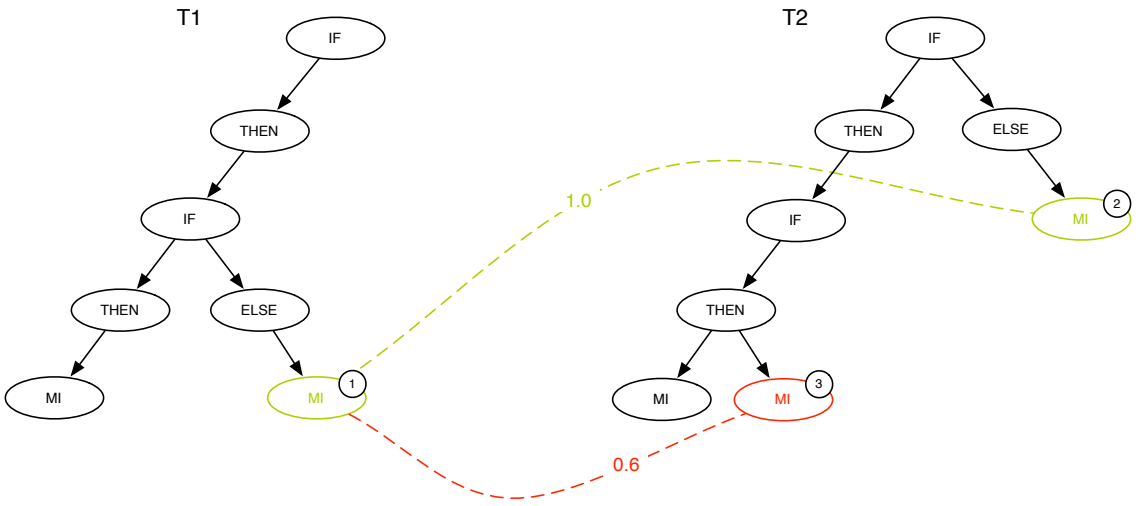


Figure 5.7: A trivial example of two trees, where the post-processing step will not be able to improve matching.

is not possible under these circumstances; the parent of Node 1 has no partner (corresponding) node in T_2. During our research on source code taken from open source projects such as ArgoUML,[1] we encountered mismatch propagations over two or three levels, *e.g.*, in nested if-then-else and try-catch statements. The levels of propagation seem to correlate with the nesting depth of, *e.g.*, if-statements or loops and the number of involved statements.

Albeit their low frequency, these propagations can have huge implications on the size of the edit script and the classification of the occurred source code changes. In the next section, we present how we overcome these inadequacies and customize the matching algorithm for detecting source code changes.

In summary, the shortcomings of the original algorithm for extracting source code changes are: 1) inadequate matching of node values, 2) using the *first* match instead of finding the *best* match, and 3) the propagation of mismatches in small subtrees. We have addressed these shortcomings and next, we present a solution to improve the extraction of source code changes.

[1] `http://argouml.tigris.org`

5.3 Change Distilling Algorithm

We motivated that the hierarchical change detection algorithm by Chawathe *et al.* needs to be adapted to take source code characteristics into account. In addition, we have discussed the circumstances under which the assumptions made for hierarchically structured text documents do not hold to compute a minimal edit script transforming an original AST into a modified AST (see Section 5.2.3). In this section, we discuss which parts of Chawathe *et al.*'s matching algorithm need to be customized for source code change extraction. Based on the desired improvements, we describe what measures and techniques overcome the inadequacy of the matching criteria discussed in the previous section.

To meet the requirements of source code change characteristics, we improve the original matching procedure with the following steps:

1. *Customize node value matching*: Since leaf matching is crucial to minimize the edit script, we aim at finding an adequate string similarity measure to match source code statements.

2. *Customize inner node matching*: We aim at finding a tree similarity measure that flexible matches inner nodes even if some unintended mismatches occur on the leaf level.

3. *Introduce best match*: Chawathe *et al.*'s Assumption 1 does not apply to source code because often multiple matching candidates for an original node are found. To address multiple matches, we select the leaf pair with the highest similarity.

4. *Use dynamic thresholds for inner node matching*: Propagation of mismatches leads to an enormous amount of unintended deletions and insertions. This is especially prominent for small subtrees—independent of the accuracy of the selected string similarity measure. Thus, we aim at finding a solution for matching small trees more adequately.

We proceed by developing similarity measures to reach the desired improvements. In the following, we discuss existing string and tree similarity measures that are adequate for source code and introduce our change distilling algorithm.

5.3.1 Matching of Leaves

Mismatches at the leaf level have tremendous impact on the size of the edit script. They can lead to mismatch propagation to higher levels in the tree and, consequently, to unnecessary node insert, delete, and move operations. String similarity measures that are robust to detect common source code changes as well as techniques to reduce the amount of false first matches are crucial to overcome mismatch propagation.

We have evaluated string similarity measures provided by SIMPACK, a generic Java library for similarities and ontologies [BKK05]. In this evaluation two measures were evident to be suitable for source code change extraction.

The Levenshtein String Similarity Measure

The Levenshtein Distance [Lev66] denotes the minimum number of operations needed to transform one string into the other. These operations are 1) insert a character, 2) delete a character, or 3) substitute a character. The algorithm is based on the problem of the *longest common subsequence*. A larger distance means that the strings are less similar, *i.e.*, that more operations are necessary to transform one string into another, whereas a distance of 0 operations denotes that the strings are equal. The runtime-complexity is $O(n \cdot m)$, where n is the number of characters in s_a and m in s_b.

For our concerns, distances are less useful than similarities, since we cannot state that a distance of 3 is generally better than a distance of 4. It depends on the lengths of the compared strings. To overcome this situation, we normalize and convert the distance, using a distance-to-similarity conversion:

$$sim_{\mathrm{Lev}}(s_a, s_b) = 1.0 - \frac{D(s_a, s_b)}{D_{\mathrm{worstcase}}(s_a, s_b)}$$

The denominator $D_{\mathrm{worstcase}}$ is equal to the maximum costs experienced under the assumption that the longest common subsequence of s_a and s_b has a length of 0, *i.e.*, that they have no characters in common: $D_{\mathrm{worstcase}} = \max(m, n)$.

The Levenshtein Distance is susceptible to changes of word or character order. Consider the strings $s_1 = verticalDrawAction$ and $s_2 = drawVerticalAction$. If they

are found at the same position in two versions of a source code entity, then it is very likely that someone has performed a refactoring, *e.g.*, by unifying identifier-nomenclature. The Levenshtein Distance does not recognize this similarity as our example illustrates: the longest common subsequence is '*verticalAction*'. The remaining characters cause four insertions and four deletions, *i.e.*, a total of eight change operations, and a distance of 8 respectively, leading to a string similarity of $sim_{\text{Lev}}(s_1, s_2) = 1 - \frac{8}{18} \approx 0.56$.

Levenshtein Distance is inadequate in this case. Since we noticed during prototyping that a lot of unintentional mismatches on the leaf-level were actually based on the deficiencies of the string similarity measure, we were eager to find an algorithm showing more robustness.

String Similarity Measures using n-grams

A family of string similarity measures is based on the *Dice Coefficient* [Dic45] | a modification of the *Jaccard Coefficient* [Jac12]. Adamson and Boreham used the Dice Coefficient to rate the similarity of strings by setting their n-grams into relation [AB74]. n-grams are bags and constructed by putting a sliding-window of length n over a string and extracting at each position the n underlying characters. For instance, the tri-grams of the string "vertical" are: $3\text{-}grams(\text{vertical}) = \{$'ver', 'ert', 'rti', 'tic', 'ica', 'cal'$\}$. The n-gram similarity measure defined by Adamson and Boreham is the ratio of twice the number of shared n-grams and the total numbers of n-grams in two strings:

$$sim_{\text{ng}}(s_a, s_b) = \frac{2 \times |n\text{-}grams(s_a) \cap n\text{-}grams(s_b)|}{|n\text{-}grams(s_a) \cup n\text{-}grams(s_b)|}$$

Dice Coefficient with bi- and tri-grams is a popular word similarity measure. In combination with source code, bi-grams have been used by Xing and Stroulia for their UMLDiff approach [XS05b], tri-grams by Weidl and Gall for their CORET approach [WG98].

To illustrate the applicability of the n-gram similarity measure for source code change detection, we calculate the similarities for strings on which the Levensthein measure fails. As before, the strings to use are $s_1 = verticalDrawAction$ and $s_2 = drawVerticalAction$. The similarities for bi-, tri-, and four-grams are: $sim_{2\text{g}}(s_1, s_2) = \frac{2\times14}{34} \approx 0.82$, $sim_{3\text{g}}(s_1, s_2) = \frac{2\times12}{32} \approx 0.75$, and $sim_{4\text{g}}(s_1, s_2) = \frac{2\times10}{30} \approx 0.67$. Using

a hash-table to store the n-grams of both strings, the runtime complexity of the n-gram similarity measure is in $O(n + m)$—one order of magnitude faster than Levenshtein.

The n-gram similarity measure is more robust to changes to the word order, since it does not rely on the longest common subsequence. It primarily focuses on common characters and secondarily on word order. Regarding source code in general and source code identifiers in particular, the measure allows a more intuitive similarity scoring. During our experiments, the measure performed worse than Levenshtein only under rare circumstances (rare in conjunction with source code): it seems to be more susceptible to substitutions including misspellings due to phonetical reasons that are common in natural language but not so in source code. The strings *Levenshtein* and *Levnshtain* for example, score with a similarity ~ 0.72 when Levenshtein is used, but only with 0.5 when bi-grams are used. Furthermore, the measure is limited to strings of a certain maximum length since the given number of different characters is finite. As a string gets longer, it will become more likely that most permutations between characters are covered. The amount of character pairs in the intersection will therefore increase, leading to an imprecise similarity. However, we were not yet able to prove this expectation experimentally, but instead, we were able to confirm the effectiveness of the n-gram similarity measure to source code on the statement-level in our evaluation (see Section 5.5).

5.3.2 Similarity Rating for Best Match

As we have discussed in Section 5.2.3, Assumption 1 does not hold for source code represented in an AST. The post-processing step proposed by Chawathe *et al.* does not succeed either. Consequently, a *first* match cannot become a *best* match using the Assumption 1 and the post-processing step.

In general, a first match that is not the best match is formalized as follows:

> Let x be a leaf in T_1 and y be its matching partner in T_2. Furthermore, let z be another leaf in T_2 and f be the threshold, so that
>
> $$sim(v(x), v(y)) \geq f \text{ and } sim(v(x), v(z)) \geq f \text{ but}$$
> $$sim(v(x), v(y)) > sim(v(x), v(z))$$

Whenever z will be visited before y during post-order traversal, a sub-optimal matching will be calculated.

Accordingly, we can derive a solution for that: Let x be a leaf in T_1. Furthermore, let mp_i be its i-th possible matching partner in T_2, such that $i \in N$ and

$$sim(v(x), v(mp_i)) \geq f$$

We mark (x, mp_i) as *best match* until we find another possible partner $mp_{i+\epsilon}$, such that $\epsilon \in N$ and

$$sim(v(x), v(mp_{i+\epsilon})) > sim(v(x), v(mp_i))$$

In this case, we mark $(x, p_{i+\epsilon})$ as *best match*. We repeat until we have tried to match all possible partners in T_2 to x.

The solution involves finding the matching partner $y \in T_2$ that matches $x \in T_1$ best. There are combinations of statements, so that x in T_1 has more than one possible partner, *e.g.*, when one and the same statement can be found over and over again in a block of code (for example print-outs for debugging). In this case, we apply the heuristics that unchanged statements stay in situ between subsequent versions of a source code entity: the first 'best' match, *i.e.*, the matching pair with the highest similarity score that has been visited during post-order traversal first, will make it into the final matching set.

So far, we have developed an approach for finding the best partner $y \in T_2$ for leaf $x \in T_1$. But this relationship is not always a two-way optimum, *i.e.*, x is not always the best partner for y. We can overcome this by calculating the similarity of each leaf pair $(x_i, y_j) \in T_1 \times T_2$ and add those pairs to the final matching set that show highest similarity.

5.3.3 Matching of Inner Nodes

Leaf matching propagates to inner nodes, as similarity on inner nodes is calculated by the number of matching leaves. A measure for inner nodes that takes leaf matching into account and is robust to potential mismatches or small subtrees is important for a maximal matching set. Chawathe *et al.* presented a simple but adequate tree similarity measure for inner nodes (Matching Criterion 2). In this section we discuss the suitability of this measure and other measures in terms of source code characteristics as well as small subtrees.

Tree Similarity used by Chawathe *et al.*

The tree similarity measure used by Chawathe *et al.* (Matching Criterion 2) takes only descending leaves into account, when deciding whether two nodes should match. Inner node descendants are ignored completely. This is an adequate approach for similarity analysis of structured text documents such as those that are written in LaTeX, where the inner nodes are used for structuring means and do not hold any semantics. For source code, inner nodes are more important, since some of them cover fundamental constructs, such as iterations as well as alternatives or exception handling.

Since, for instance, an else-block may contain an if-statement, matching between descendants can occur. During our studies, it happened that an else-block matched with a descendant else-block. This matching resulted in a none applicable move operation, since a parent node cannot become a child node of one of its descendants. To overcome such situations, we added the check that the string similarity of the value of inner nodes must also satisfy the threshold t. Whenever a node does not have its own value, it inherits that of its parent to emphasize their affiliation.

Dice Coefficient for Inner Nodes

By using the Dice Coefficient, we get a measure taking inner nodes into account. In conjunction with code clone detection, Baxter *et al.* used the Dice Coefficient to calculate the similarity of two ASTs [BYdM+98]. For our purpose, we apply the same measure to inner nodes:

$$sim_{\text{Dice}}(T_a, T_b) = \frac{2 \times |nodes(T_a) \cap nodes(T_b)|}{|nodes(T_a) \cup nodes(T_b)|}$$

where $nodes(T_x)$ denotes all nodes of T_x including the root.

Taking inner nodes of the subtrees into account does not impact the value of the similarity measure, because the matching of leaves propagates to inner nodes. A more important aspect of the Dice Coefficient is that common nodes of T_a and T_b are weighted more than mismatches. When two trees share most of their nodes, but T_b differs in structure from T_a by few changes, the Dice Coefficient is more robust than the measure used by Chawathe *et al.* Overall, our evaluation

showed that the algorithm of Chawathe *et al.* including inner node similarity weighting and dynamic threshold (see next sections) performs better than the Dice Coefficient.

Inner Node Similarity Weighting

According to our adapted Matching Criterion 2, the similarity of inner node values and the similarity of their subtrees influence the similarity for inner nodes likewise. Therefore, two inner nodes do not match either because their node values mismatch or they have too few leaves in common. Regarding if-statements or loops, a value, *i.e.*, condition expression, mismatch may cause a tremendous amount of unnecessary changes. We overcome this situation by weighting the common leaves function more than the similarity of values between inner nodes.

Inhibiting Propagation of Mismatches in Small Subtrees

The similarity measures for strings as well as for trees introduced in the previous sections reduce mismatching of single nodes, but do not reduce them for small subtrees. Consider the code snippets in Figure 5.8. According to Matching Criterion 2, the similarity between the two then-blocks of the if-statements is 0.5 (one shared node, two leaves) causing a mismatch of the then-blocks and the if-statements.

To weaken the high impact that small changes can have on small subtrees, we dynamically lower thresholds for small subtrees; dynamically, meaning in regard to the size of the subtrees under investigation. We experienced adequate matching results for $t = 0.6$ if $n > 4$ and $t = 0.4$ if $n \leq 4$, where n is the number of leaf-descendants of the inner node.

Lowering thresholds for all inner nodes, no matter how many leaf-descendants they count, injects undesired behavior into the algorithm: the amount of similar inner nodes increases by lowering the threshold leading to false matches.

5.3.4 Our Matching Algorithm Used for Change Distilling

In this section we present we present our improved tree differencing algorithm suitable to extract changes in source code. To recall, our improvements are:

```
if (cancelled()) {
   close();

}
```
(a)

```
if (cancelled()) {
   close();
   logger.debug("user has cancelled action");
}
```
(b)

Figure 5.8: (a) A small if-statement; (b) A logging statement has been added.

1. Using bi-grams as a robust string similarity measure that is able to cover common changes of source code identifiers.
2. Adding a similarity check of node values to Chawathe *et al.*'s tree similarity measure to solve the problem of descendant subtree matching.
3. Using inner node similarity weighting to reduce inadequate mismatches of condition expressions.
4. Introducing the best match algorithm to reduce the impact of Chawathe *et al.*'s Assumption 1.
5. Using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

We evaluated combinations of the discussed string and tree similarity measures as well as best match, dynamic threshold, and inner node similarity weighting with our benchmark (see Section 5.5 for a detailed discussion). The following combination of measures and techniques performed best for extracting source code changes:

- For Matching Criterion 1 (Leaves) we use the bi-gram string similarity measure:

$$match_1(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

  where $f = 0.6$.

- In addition to Matching Criterion 1, we take the *best* match for a leaf x instead of the *first* match.

- Matching Criterion 2 (Inner nodes) is extended by the check whether the values of the inner nodes are similar:

$$match_2(x,y) = \begin{cases} true & \text{if } l(x) = l(y) \ \wedge \\ & \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \ \wedge \\ & sim_{2g}(v(x), v(y)) \geq f \\ false & \text{otherwise} \end{cases}$$

  where $f = 0.6$ and $t = 0.6$.

- We add inner node similarity weighting: if the string similarity of inner node values, *e.g.*, the condition of an if-statement, is less than the threshold f, but $\frac{|common(x,y)|}{max(|x|,|y|)} \geq 0.8$ holds, $match_2(x,y)$ is true.

- The threshold for the inner node similarity measure is adjusted dynamically for small subtrees: $n \leq 4 \rightarrow t = 0.4$.

The final algorithm is presented in Figure 5.9. The input to the algorithm are two labeled and valued trees T_1 and T_2. The algorithm first calculates a complete matching of all leaves (Lines 5–9). The leaf pairs are sorted (Line 10) according to their similarity and the best matches are added to the final matching set (Lines 11–15). At the end, the inner nodes are matched using dynamic thresholds (Lines 17–22). The output of the algorithm is a set of matching node pairs that is used by the edit script algorithm to compute the tree edit operations.

The runtime analysis of the matching algorithm from Chawathe *et al.* has to be extended by the additional computation steps. Assume $n = \max(|T_1|, |T_2|)$, where $|T|$ is the number of leaves. The costs to compare two leaves is denoted by c. The matching of all leaves is in $O(n^2 c)$, *i.e.*, $O(n^2)$, since we have to compare each possible leaf pair. Sorting the generated $O(n^2)$ matching pairs is in $O(n^2 \log n^2)$. For each pair that is added to M_{final}, the whole M_{tmp} has to be traversed at most once to remove all corresponding leaf pairs. Thus, building M_{final} for the leaves is proportional to $n^2(1 + c + \log n^2)$. The runtime complexity of inner node matching can be derived from the original work by Chawathe *et al.*: the number of inner nodes in T_1 and T_2 is denoted by m. Matching Criterion 2 can be computed for all inner nodes in $O(mn)$ (we refer to [CRGMW96] for more details). In addition, the value comparison of the inner nodes is in $O(mc)$. The overall runtime of

1: **Input**: trees T_1, T_2
2: **Result**: final matching set: M_{final}
3: $M_{\text{final}} \leftarrow \phi$, $M_{\text{tmp}} \leftarrow \phi$
4: Mark all nodes in T_1 and T_2 "unmatched"
5: **for all** leaf $x \in T_1$ and leaf $y \in T_2$ **do**
6: **if** $match_1(x, y)$ **then**
7: $M_{\text{tmp}} \leftarrow M_{\text{tmp}} \cup (x, y, sim_{2g}(v(x), v(y)))$
8: **end if**
9: **end for**
10: Sort M_{tmp} into descending order, according to the leaf-pair-similarity
11: **for all** leaf-pair-similarity $(x, y, sim_{2g}(v(x), v(y))) \in M_{\text{tmp}}$ **do**
12: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
13: Remove all leaf-pairs from M_{tmp} that contain x or y
14: Mark x and y "matched"
15: **end for**
16: Proceed post-order on trees T_1 and T_2:
17: **for all** unmatched node $x \in T_1$, if there is an unmatched node $y \in T_2$ **do**
18: **if** $match_2(x, y)$ (incl. dynamic threshold and inner node similarity weighting) **then**
19: $M_{\text{final}} \leftarrow M_{\text{final}} \cup (x, y)$
20: Mark x and y "matched"
21: **end if**
22: **end for**

Figure 5.9: Our Matching Algorithm used for Change Distilling

inner node matching is $O(m(c + n))$. In summary, the total time of the matching algorithm is proportional to:

$$n^2(c + 1 + \log n^2) + m(c + n)$$

Compared to the original algorithm by Chawathe *et al.*, our runtime is $O(\log n^2)$ slower. We describe in the next section how we mitigate the impact of this additional factor in order to optimize the runtime performance of our change distilling algorithm.

5.4 Implementation

We built the Eclipse plugin CHANGEDISTILLER that implements our change distilling algorithm. Our current implementation relies on the CVS capabilities, Java Development Tools (JDT),[2] and compare functionality of Eclipse. The extracted source code changes are stored in a Hibernate[3] mapped database.

We have automated the process of change distilling within Eclipse. Starting with an Eclipse project, CHANGEDISTILLER is able to extract changes from the version chain of a single class, packages, or a whole project.

5.4.1 Fine-Grained Change Extraction Process

Figure 5.10 depicts the change extraction process of CHANGEDISTILLER. From a project under CVS control, revisions of Java classes are checked out using the CVS capabilities of Eclipse. For two subsequent revisions of a Java class we use the compare plugin to extract the methods and attributes that have changed. This pre-filtering step leads to smaller trees for comparison. Assume a class has about 1,000 lines of code, but only a single method with 20 lines of code has changed. Using the compare plugin[4] reduces the input to our change distilling algorithm significantly. Recalling the runtime complexity of the matching algorithm, this is a considerable performance gain, as the input trees are kept small.

For both versions of a changed method or attribute intermediate ASTs are created using the AST visitor from JDT. Creating intermediate trees is necessary, since the matching algorithm expects labeled and valued nodes as well as a uniquely defined parent child relationship between hierarchically situated nodes. This expectation is not covered by ASTs created by JDT. For instance, an if-statement may have two children—a then- and an else-block. Depending on the AST implementation, the access from the if-statement (parent) to the two blocks (children) is not available through "getChildren," but through "getThenBlock" and "getElseBlock." Leaves in the intermediate AST are normal statements, with the statement kind as label and the statement itself as value. For instance the leaf of statement

[2] `http://www.eclipse.org/jdt`

[3] `http://www.hibernate.org`

[4] The complexity of the compare plugin is in $O(n^2)$ where n is the number of members of a Java class.

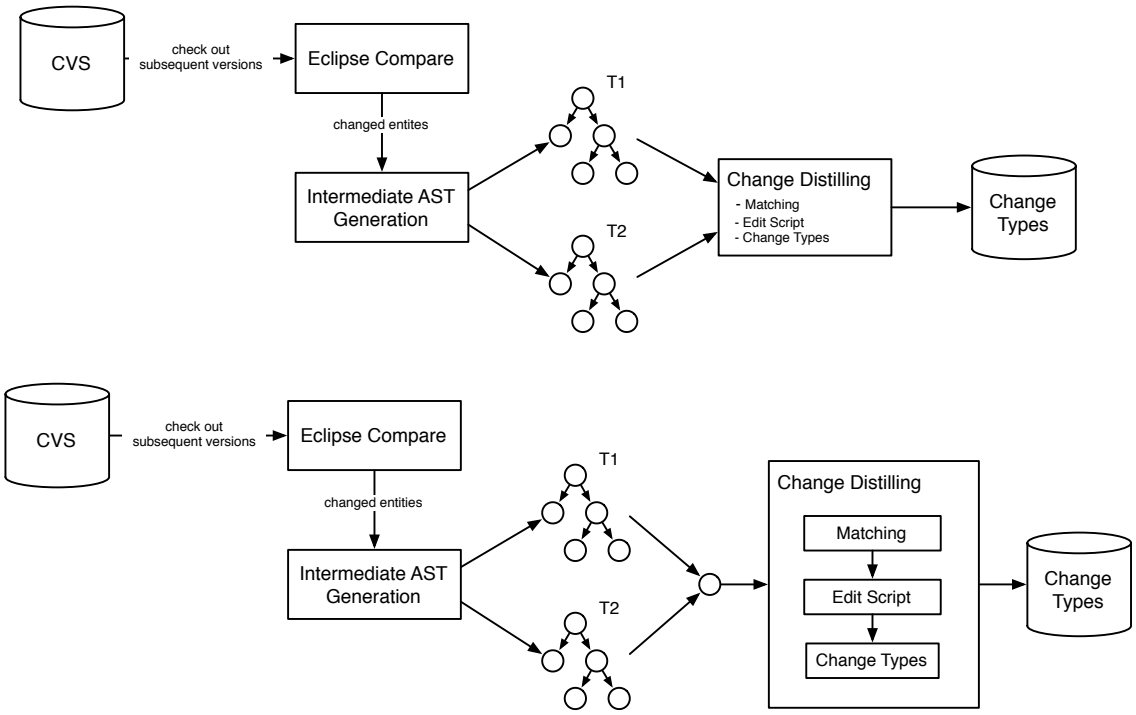


Figure 5.10: Fine-grained Change Extraction Process

`foo.bar();` has the label *MI* and the value *"foo.bar();"*.

The intermediate ASTs T_1 and T_2 are then fed into our change distilling algorithm. The algorithm can be configured with different string and tree similarity algorithms and thresholds as described in Section 5.3. The output is a set of basic tree edit operations that are classified to change types and stored into the Hibernate mapped database.

5.4.2 Classifying Tree Edit Operations

In [FG06] we have assigned basic tree edit operations to change types. For instance, the tree edit operation *Statement Ordering Change* is $\text{MOV}(s, p(s), k)$, meaning that the statement s is moved to position k in the children of its parent $p(s)$.

Sometimes, we can infer that an update took place even if the similarity of the two strings under comparison is too low. Consider the two methods `foo(Object myParam)` in revision $n-1$ and `foo(Figure myParam)` in revision n. A parameter type change from "Object" to "Figure" happened but the similarity

of the two strings "Object" and "Figure" is below the threshold $f = 0.6$ hence is not matched. Therefore, by classifying the tree edit operation without any further check, a new parameter would be inserted and an old one would be deleted. Since the parameter name did not change, the classifier is able to classify the two operations as a *Parameter Type Change* by checking whether the parents of "Object" and "Figure" are equal.

5.5 Evaluation

In Section 5.3.4, we have described our change distilling algorithm. To investigate the quality of our improvements, we developed an extensive benchmark. The benchmark consists of a set of special test cases and of a large data set of manually classified changes. The data set is taken from three different open source case studies: ArgoUML,[5] Azureus,[6] and JDT[7]. With the benchmark, we show that our improvements approximate the minimum conforming edit script more closely than Chawathe *et al.*'s change detection algorithm. Although the CHANGEDISTILLER is able to detect changes on the class level as well, our benchmark focuses on changes on the method level. Since our major interest lies in the tree differencing part of our algorithm, changes on the method level are sufficient—they cover all tree structures that may occur in an AST.

5.5.1 Preliminaries

The final step of CHANGEDISTILLER is to analyze, consolidate, and classify the tree edit operations into *change types* [FG06]. Change types are the most suitable data set for benchmarking our change distilling algorithm, because they are an adequate measure for the quality of our algorithm as well as straightforward to implement and validate manually. Change types represent the kind of changes that a human will intuitively find when she compares two subsequent versions of a Java method. For example, she will recognize that a *method invocation* has been inserted into a method, rather than thinking of the corresponding tree edit operation.

[5] `http://www.argouml.org`
[6] `http://azureus.sourceforge.net`
[7] `http://www.eclipse.org/jdt`

Taking two versions ($n-1$, n) of a Java method, we count the occurrences of each particular change type manually. We, then, run the CHANGEDISTILLER on the same pair of versions. For each version pair ($n-1$, n) and each change type t, we calculate the mean absolute error ϵ_t and the mean absolute percentage error δ_t:

$$\epsilon_t = \frac{1}{k}\sum_{i=1}^{k}|x_i(t) - \tilde{x}_i(t)|, \quad \delta_t = \frac{1}{k}\sum_{i=1}^{k}\left|\frac{x_i(t) - \tilde{x}_i(t)}{x_i(t)}\right|$$

where $x_i(t)$ is the expected number of occurrences of change type t, $\tilde{x}_i(t)$ the found number of occurrences of change type t, and k the number of version pairs in which t was expected or found. The smaller the difference between the number of change types classified manually and found by CHANGEDISTILLER, the smaller the error and the better we consider the performance of our algorithm.

For each version pair ($n-1$, n), we calculate the mean absolute error ϵ and the mean absolute percentage error δ for the edit script:

$$\epsilon = \frac{1}{m}\sum_{i=1}^{m}|x_i - \tilde{x}_i|, \quad \delta = \frac{1}{m}\sum_{i=1}^{m}\left|\frac{x_i - \tilde{x}_i}{x_i}\right|$$

where x_i is the expected length of the edit script, $\tilde{x}_i$ the found length of the edit script, and m the number of version pairs.

Before applying these measures to our change distilling algorithm, we have to discuss one shortcoming in terms of counting change types for the benchmark: we cannot evaluate exactly, where the change occurred, since we do not store its exact location in the benchmark, but rather in which version and method it was found. This means that we can tell that, *e.g.*, two *statement inserts* were found in method `foo()` between Version 1.11 and 1.12, but not whether the statements were, for example, inserted into a particular then-block or somewhere else. Performing a manual qualitative analysis on the whole data set, instead of restricting ourselves to a quantitative evaluation, is barely feasible; we would have to determine the exact location in the AST for each change by hand in order to compare it to the output of our algorithm. For a sufficiently large set of changes, this is too time consuming and error prone.

To show that counting the occurrence of change types is sufficient nonetheless,

we performed a qualitative evaluation on a randomly selected sample of the data in our benchmark. For this, we have calculated precision and recall as follows:

$$Precision = \frac{\text{\# relevant changes found}}{\text{\# changes found}}$$

$$Recall = \frac{\text{\# relevant changes found}}{\text{\# changes expected}}$$

The selected sample contains 13 pairs of Java method versions comprising 120 expected changes. We compared each of the 151 changes found by CHANGEDISTILLER with the expected changes manually and obtained a precision of $\frac{118}{151} = 0.78$ and a recall of $\frac{118}{120} = 0.98$. Furthermore, we observed that the found edit scripts always transform the old into the new version of the Java methods correctly. Consequently, a recall < 1.0 denotes that our algorithm found changes that replace the ones that we expected. For instance, the method invocation
`mParameter.setKind(MParameterDirectionKind.IN)`[8]
was updated with
`ModelFacade.setKindToIn(mParameter)`,
but CHANGEDISTILLER found a corresponding *statement delete* and *statement insert* instead. A precision < 1.0 denotes that our algorithm found a non-minimal conforming edit script with *virtual* changes, *i.e.*, pairs of changes in the same edit script of which the second reverts the first one and vice-versa. Consider the following concrete example of source code in Figure 5.11 taken from our benchmark. For this, we manually classified four statement inserts (one if-statement insert and three method invocations). For this particular case, our change distilling algorithm extracts five statement inserts, one statement delete, and two statement parent changes leading to an absolute error ϵ of 4 and a percentage error δ of 50% of the length of the edit script. Since the top-most if-statements (Line 1) share only two out of five leaves (Line 2 and 3 in (a) with Line 2 and 8 in (b)), Matching Criterion 2 is not satisfied, *i.e.*, they do not match. Therefore, the edit script contains the insert and delete operations of the top-most if-statement as well as move operations of the first and the last statement from the deleted to the re-inserted if-statement.

[8] In method `addOperation(...)` of class `org.argouml.uml.reveng.java.Modeller` between Revision 1.45 and 1.46.

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(fElements, type);
4     return;
5  }
```

(a)

```
1  if (matches.length == 0) {
2     fElements =
3        growAndAddToArray(fElements, type);
4     if(SelectionEngine.DEBUG){
5        System.out.print(
6           "SELECTION - accept type("
7        );
8        System.out.print(type.toString());
9        System.out.println(")");
10    }
11    return;
12 }
```

(b)

Figure 5.11: (a) The original if-statement; (b) The modified if-statement: of method `acceptSourceMethod(..)` of class `jdt.internal.core.SelectionRequestor`

Applying these four changes does not transform the source code but leads to a non-minimal conforming edit script.

Regarding the high recall, we claim that our algorithm at least finds the changes we expect. But in certain cases, it finds a conforming edit script that is not minimal. If it finds fewer than expected changes such as, for example, statement updates, a set of corresponding changes are found instead (*e.g.*, in case of statement update: statement insert and delete).

With our benchmark, we show that the output of our change distilling algorithm approximates the minimum conforming edit script more closer than Chawathe *et al.*'s algorithm. Therefore, we only benchmark with the error measures.

5.5.2 Our Benchmark for Change Distilling

For the benchmark we use a combination of dedicated test cases and data from three different case studies. We discuss how we have chosen the data and what preparation steps they have undergone.

Test Cases

The test cases serve as validation for our improvements. We focused on testing string similarity measures, matching of small subtrees, and special issues on ordering changes. Test cases that failed with the original algorithm had to pass with the customized algorithm. For that, we have hard-coded exact tree edit operations and their classification between two source code version of one class. For an in-depth discussion of these test cases we refer to [W06].

Collecting Changes from Existing Software

Special test cases are well-suited to investigate specific or theoretical issues. They are insufficient for claiming whether an approach applies to real world problems or not. Therefore, we decided to integrate data from the open source projects ArgoUML (∼1,500 classes, ∼272 kLOC), Azureus (∼2,300 classes, ∼432 kLOC), and JDT of Eclipse (∼1,100 classes, ∼388 kLOC). Choosing representative test data among the about 4,900 classes was a challenge. We fed the projects into CHANGEDISTILLER with the original change extraction configuration and applied the following criteria to find appropriate Java classes.

- A lot of changes over time, few changes between revisions: We preferred classes that have 100 to 200 revisions and contain methods that show 10 to 20 changes per revision.
- Method size: We have chosen methods with 50 to 500 lines of code.
- Nesting: Methods that have nested if- and loop-statements are most interesting in terms of the small-subtree-problem.
- Diversity of changes: We preferred classes with different change types, since we want to benchmark our algorithm in a broad variety of source code structures.

According to the above criteria, we located eight candidate methods in total—each one in a different class—that we integrated into our benchmark. We performed a checkout of every revision in which the selected methods experienced changes. Preparation of the classes was done by deleting all fields and methods except the chosen ones. During manual inspection, we finally classified 1,064 changes in a total of 219 revisions. To reduce evaluator bias, two of the authors of this paper have classified the changes independently and consolidated their findings.

5.5.3 Results and Discussion

In Section 5.3 we claimed that our algorithm is better suited for source code changes than the original algorithm by Chawathe *et al.* In this section we present and discuss selected comparisons between different configurations of our change distilling algorithm, *i.e.*, we show how the different configurations perform against each other. We benchmark different combinations of:

- The original *first* match algorithm for leaves, or our *best* match algorithm.
- Either the tree similarity measure suggested by Chawathe *et al.* or the Dice Coefficient are used for inner node comparisons.
- We dynamically lower the threshold t for inner nodes to 0.4 whenever the left and the right tree roots have four or fewer descendants.
- We either turn on or off inner node similarity weighting.
- We use either Levenshtein or n-grams similarity measure to match node values.

For the string similarity measures, we use f as the threshold variable, and t as inner node similarity threshold.

Benchmarking

We have conducted four runs with different configurations:

(a) Chawathe *et al.*'s original algorithm, Levenshtein as string similarity measure, $f = 0.7$ and $t = 0.6$, dynamic thresholds and inner node similarity weighting disabled.

(b) Chawathe *et al.*'s original algorithm, bi-grams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds and node similarity weighting disabled.

(c) Our best match, bi-grams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds and inner node similarity weighting disabled.

(d) Our best match, bi-grams as string similarity measure, $f = 0.6$ and $t = 0.6$, dynamic thresholds and inner node similarity weighting enabled

The minimum conforming edit script comprises 1,064 changes and the smaller the mean absolute error ϵ and the mean absolute percentage error δ, the better the performance of the algorithm. Tables 5.1 and 5.2 depict the results from Runs (a), (b), (c), and (d) in the respective columns. Additionally, we provide more detailed tables for each run including root mean squared absolute error and root mean squared percentage error in Appendix 5.8.

Run (a) In the first run, we found fewer *statement updates* and *condition expression changes* than expected with a mean absolute error ϵ of 0.96 and 1.02 between each pair of versions. In other words, the algorithm has missed on average approximately one statement update and condition expression change per pair of versions. As indicated by the ϵ values of statement inserts and deletes, the missed statement update and condition expression change are replaced by a pair of statement inserts and deletes. The accuracy of finding *statement updates* depends on the accuracy of the string similarity measure. The fewer *statement updates*, the more *statement insert* and *deletes* are found. Besides the string similarity measure, the accuracy of finding *condition expression changes* relies on the matching of inner nodes. Two if-statements match if their conditions (*i.e.*, values) match and if the inner node similarity satisfies the threshold t. Thus, matching small trees has an impact on condition expression changes. A mismatch leads to deletes of if-statements and alternative parts with additional insert and ordering/parent changes. On the other hand, when their conditions do not match but their subtrees, a mismatch occurs as well. The original algorithm is not able to match nodes

		(a)			(b)		
Change Type	x	$\tilde{x}$	ϵ	δ	$\tilde{x}$	ϵ	δ
Alternative Part Del.	9	32	1.04	0.08	28	1.17	0.11
Alternative Part Ins.	15	40	0.86	0.06	36	0.88	0.06
Cond. Exp. Change	91	51	1.02	0.58	64	0.89	0.44
Method Renaming	1	1	0	0	1	0	0
Param. Delete	9	12	0.43	0.07	12	0.43	0.07
Param. Insert	16	20	0.29	0.04	20	0.29	0.04
Param. Ord. Change	0	19	2.71	0	19	2.71	0
Param. Renaming	3	1	0.67	0.67	2	0.75	0.5
Param. Type Change	1	1	0	0	0	1	1
Return Type Change	1	1	0	0	1	0	0
Return Type Insert	1	1	0	0	1	0	0
Stmt. Delete	144	371	2.28	0.3	283	1.96	0.29
Stmt. Insert	391	640	2.15	0.4	552	1.89	0.42
Stmt. Ord. Change	14	105	2.26	0.12	82	2.23	0.19
Stmt. Parent Change	86	185	1.84	0.2	194	1.87	0.21
Stmt. Update	282	216	0.96	0.34	318	0.72	0.19
Runtime		$\sim 12\,\mathrm{s}$			$\sim 5\,\mathrm{s}$		
Total	1064	1696	3.27	0.79	1613	2.91	0.72

Table 5.1: Benchmark results of the Runs (a) and (b) including the run-time performance in seconds, ϵ as well as δ per change type and edit script for each configuration.

accurately, leading to a mean absolute percentage error δ of 0.79 with additional 3.27 changes per version pair as depicted in Column (a).

Run (b) While evaluating the results of the initial run, we found that the outcome mainly relies on the string similarity measure as well as on the chosen threshold. We, therefore, lowered the threshold to $f = 0.6$ and used bi-grams as string similarity measure instead of Levenshtein. The Column (b) of Table 5.1 illustrates that the number of statement updates increased tremendously compared to the number of condition expression changes—it even exceeded the expected number of statement updates. The reason for this increase is the flexibility of the bi-gram similarity measure, leading to statement updates instead of inserts and deletes. Configuration (b) reduced the overall ϵ from 3.27 to 2.91. This decreased the δ by 7% down to 72%.

		(c)			(d)		
Change Type	x	$\tilde{x}$	ϵ	δ	$\tilde{x}$	ϵ	δ
Alternative Part Del.	9	25	1.06	0.12	14	0.78	0.22
Alternative Part Ins.	15	33	0.78	0.07	22	0.41	0
Cond. Exp. Change	91	58	0.92	0.47	85	0.58	0.24
Method Renaming	1	1	0	0	1	0	0
Param. Delete	9	11	0.33	0.08	11	0.33	0.08
Param. Insert	16	19	0.23	0.04	19	0.23	0.04
Param. Ord. Change	0	19	2.71	0	17	3.4	0
Param. Renaming	3	1	0.67	0.67	1	0.67	0.67
Param. Type Change	1	1	0	0	1	0	0
Return Type Change	1	1	0	0	1	0	0
Return Type Insert	1	1	0	0	1	0	0
Stmt. Delete	144	264	1.84	0.32	225	1.77	0.34
Stmt. Insert	391	533	1.76	0.38	494	1.54	0.32
Stmt. Ord. Change	14	83	2.08	0.11	73	2.23	0.08
Stmt. Parent Change	86	162	1.56	0.11	118	1.14	0.21
Stmt. Update	282	259	0.41	0.14	260	0.41	0.14
Runtime			~ 9 s			~ 9 s	
Total	1064	1471	2.2	0.52	1343	1.64	0.34

Table 5.2: Benchmark results of the Runs (c) and (d) including the run-time performance in seconds, ϵ as well as δ per change type and edit script for each configuration.

Run (c) To further improve the result, in particular, to reduce the number of statement updates, we used our best match algorithm with bi-grams. Column (c) of Table 5.2 shows the corresponding results. Using the best match algorithm reduces the number of statement updates and increases the condition expression changes. The advantage of best match is that it is less likely that correct statement inserts/deletes are replaced by updates, because better matches are taken for the matching set. Using best match improved the output of the algorithm significantly. We achieved a δ of 52%, thus we further reduced the ϵ by 0.71 to 2.2.

Run (d) The results of the last and most influencing improvement, *i.e.*, our matching algorithm, are shown in Column (d) of Table 5.2. In particular, the *inner node similarity weighting* and *dynamic threshold* increased the number of condition expression changes. The number of statement inserts, deletes, and ordering changes as well as the alternative part inserts and deletes were reduced. The

reason for the decrease of those changes was that more if-statements matched and therefore fewer statements were moved to a new if-statement.

Using the dynamic thresholds, we are able to get rid of the mismatch propagation in small subtrees. This led to an improvement of the overall δ by 8%. Enabling the weighting of the inner node similarity derived a further decrease of the δ by 10%.

Concerning the runtime, we observed a decrease between the Runs (a) and (b) as well as an increase between (b) and (c) or (d). The Levenshtein similarity measure used in Run (a) is an order of magnitude slower than the bi-gram similarity measure used in Run (b). The best match algorithm used in Run (c) and (d) is slower than first match used in Run (a).

Our change distilling, in particular the configuration we used in Run (d), reduced the mean absolute percentage error δ by 45% from 79% to 34% compared to the original algorithm. The number of additional changes found was reduced by 2.08 from 3.27 to 1.64 per pair of versions.

Further Benchmark Runs

We performed further benchmarking using the Dice Coefficient and other n-grams. We do not discuss these results in detail, as they were not as promising as our configuration used in Run (d), but summarize them briefly; using tri- or four-grams instead of bi-grams resulted in an δ of 38% and 40%. Since tri- and four-grams are less flexible than bi-grams, fewer statement updates occurred. The Dice Coefficient for inner node matching combined with the various configurations resulted in a minimum ϵ of 43% which is lower than the one that was achieved with the inner node similarity of Chawathe *et al.*.

5.5.4 Limitations

Coming back to the results in Tables 5.1 and 5.2, our algorithm is still limited in finding the appropriate number of move operations. In particular, the performances of *parameter ordering changes* and *statement ordering changes* are modest. After an in-depth inspection of the benchmark results, we found that the method `acceptSourceMethod(...)`[9] was responsible for these outliers. Removing this method from the benchmark yielded a δ of 30%; this is a further improvement of 4%. The number of parameter changes was decreased to one and all declarations changes were extracted correctly.

Concerning body changes, the main reason for the few additional errors was also due to this method because it mainly consists of small nested if- and loop-statements. Although we used our dynamic threshold approach, these small blocks were not matched because 1) the node similarities of those blocks fall below 0.4, and 2) the depths of their subtrees are mostly bigger than 4.

Furthermore, the best match approach may match reoccurring statements that are not at the same position in the method body. For instance, consider that the first statement of a method changed, but the same statement reoccurs at the end of the method and stays unchanged. The best match approach will match the first with the last statement leading to a mismatch for the first statement. Such a mismatch can have, as in this particular case, tremendous impact on the extraction of other changes. We noticed that such mismatches led to replacements of nested if- and loop-statements. Currently we are investigating post-processing steps that take the position of statements into account to remove inappropriate matches.

The declaration changes, in particular, the parameter ordering changes are also an implication of the small tree problem. The parameter changes in Figure 5.12 happened from Revision 1.35 to 1.39 of the `acceptSourceMethod(...)` described above; three new parameters were inserted. The similarity between the parameter-list nodes is 0.57 ($\frac{4}{7}$), thus the nodes do not match. This mismatch yields to the changes: 1) deletion of the old parameter list, 2) insertion of a new parameter list, 3) insertion of the three new parameters, and 4) move of the existing parameters to the new list. Besides the three parameter insertions, four additional parameter ordering changes are classified—the parameter list insert and delete are omitted.

As we have selected the methods for the benchmark randomly and the δ of our

[9] in `org.eclipse.jdt.internal.core.SelectionRequestor`

```
protected void acceptSourceMethod(
    IType type,
    char[] selector,
    char[][] parameterPackageNames,
    char[][] parameterTypeNames)
```
(Revision 1.35)

```
protected void acceptSourceMethod(
    IType type,
    char[] selector,
    char[][] parameterPackageNames,
    char[][] parameterTypeNames,
    boolean isDeclaration,
    int start,
    int end)
```
(Revision 1.39)

Figure 5.12: Parameter changes in between Revisions 1.35 and 1.39 of the method `acceptSourceMethod(...)`.

algorithm is for all methods about 30%, except for the method described above, we claim that unsolvable small tree problems occur relatively seldom. However, further investigations of this issue are needed and subject to future work.

5.5.5 Summary

To validate our improvements, we established an extensive benchmark comprising 1,064 manually classified changes. Compared to the original algorithm of Chawathe *et al.*, we approximate the minimum conforming edit script with a mean absolute error of 1.64 and a mean absolute percentage error of 34% per version pair, *i.e.*, an improvement of 45%. This means, on average, we find less than two additional change types, whereas the original algorithm finds more than three additional change types between two versions. The results showed that the combination of our best match algorithm with bi-grams, Chawathe *et al.*'s node similarity measure, dynamic thresholds, and the inner node similarity weighting achieved the best benchmark results.

Although our dynamic thresholds inhibit mismatch propagation in small subtrees noticeably, we consider the problem as not fully solved yet, as the changes in method `acceptSource-Method(...)` showed.

5.6 Related Work

Kim and Notkin [KN06] pointed out that source code differencing has proven itself to be a long term research topic fundamental to multi-version program analyses. Existing approaches either rely on lexical, syntactical, or semantical differencing techniques. A further classification can be done with respect to the granularity of the algorithms, *i.e.*, whether they perform coarse-grained or fine-grained change extraction as well as analyses. Our algorithm identifies fine-grained syntactical changes. In [HH04] Hassan and Holt propose evolutionary code extractors in general. They discuss the need of such tools as well as the level of source code extraction granularity.

The algorithm presented by Chawathe *et al.* as well as our change distilling algorithm are closely related to tree differencing in general and to the tree edit distance problem in particular. The tree edit distance problem is to compute the edit distance based on a corresponding edit script between two labeled ordered or unordered trees [Bil05]. The edit operations used are: 1) change the label of a node (relabel), 2) delete a non-root node, and 3) insert a node. One of the first non-naive algorithms for the tree edit distance problem was introduced by Tai [Tai79]. The quadratic upper bound of this general approach has been improved by Shasha and Zhang [SZ90, Zha95]. These algorithms are inappropriate for our concerns: 1) they do not act on labeled, ordered, and valued trees, 2) the operation *relabel* cannot be used for source code, as, for instance, a method invocation must not become an assignment, 3) they do not support move operations, and 4) do not support updates of values. The algorithm of Chawathe *et al.* addresses these issues and, additionally, is faster than these general tree edit distance algorithms.

Existing differencing tools such as the well-known *GNU diff* [HS77] deal with flat, rather than with hierarchical information. They are usually based on the *longest common subsequence* (LCS) algorithm and calculate textual changes, *i.e.*, a list of lines that were changed, inserted, or deleted. *GNU diff* cannot for example distinguish between changes applied to license information or documentation and changes applied to a method body. In contrast to *diff*, our algorithm can detect changes more precisely and is able to assign a particular change to a concrete source code entity (such as the declaration or body part of a method), rather than just to a line number.

In [MC04] Maletic and Collard present a language independent approach for detecting syntactic differences between source files using an intermediate representation of the source code in XML. The output provided by *GNU diff* is mapped to an XML representation to locate changed entities. Our approach does not rely on textual differences and is able to detect changes due to move operations. Recently, Canfora *et al.* reconstruct changes from differencing results provided by CVS or Subversion *diff* to track the evolution of source code lines [CCP07]. For that, they use Vector Space Models and the Levenshtein string similarity measure.

Yang describes an algorithm based on a branch-and-bound implementation of the largest common subtree problem [Yan91]. The output of the algorithm are sets of matching and modified abstract syntax tree nodes, but it is not reported what operations transform the original into the modified tree.

Horwitz's approach computes semantic as well as textual differences between two programs [Hor90]. The approach partitions a program according to its behavior extracted from the program representation graph. Similar to our approach, Horwitz builds a matching set between such partitions to extract the differences. The approach is limited to programs written in a language that supports a subset of traditional programming languages. Furthermore, our approach provides a more complete set of tree edit operations and additionally classifies changes into change types. The algorithm presented by Jackson and Ladd reports semantic changes in procedural programs [JL94]. They analyse the input-output behavior of two procedures to detect changed behavior.

Apiwattanapong *et al.* [AOH07] use enhanced control-flow graphs to model semantic behavior of methods of object-oriented programs. Identifying modified and unmodified methods is based on graph isomorphism. Their discussion of the impact of path changes caused by exception handling can be used to extend our work. Furthermore, we claim that both approaches, the one presented by Apiwattanapong *et al.* and our work, are complementary and that semantic differencing can be used to extend and refine our work.

Raghavan *et al.* implemented *Dex* [RRL+04] a tool for extracting changes between C source files. They use change information provided by patch files, to locate the changed parts in source files. These parts are fed into their tree differencing algorithm that outputs the edit operations. *Dex* can be used with our taxonomy to classify source code changes in C programs.

Tu and Godfrey used their *BEAGLE* tool to detect structural evolution of

software systems [TG02]. With origin analysis *BEAGLE* detects old functions as the "origin" of new ones based on software metrics and code clone detection. Origin analysis was also used to detect merging and splitting [GZ05] and method renaming [KPEJW05].

Recently, Xing and Stroulia presented their *UMLDiff* tool in [XS05b]. *UMLDiff* tracks changes on the interface (logical design) of classes. In contrast to our work, they are able to track when entities are moved among different classes. However, *UMLDiff* focuses on *recovering higher-level design knowledge evolution*, *i.e.*, changes on the interface level, whereas our work additionally allows fine-grained differencing on the implementation-level, *i.e.*, changes on single statements inside of method-bodies. Similar to *UMLDiff*, *SiDiff* by Kelter *et al.* extracts differences between UML models [KWN05]. The models are stored in XMI files. They use a combined top-down and bottom-up approach for matching model parts. The matching is then used to classify differences of UML models.

Kim *et al.* presented an approach to automatically infer likely changes at or above the method level [KNG07]. They use a simple matching algorithm with the Levenshtein string similarity measure. Compared to *UMLDiff* or *SiDiff*, Kim *et al.* inference approach represents the changes concisely as first-order relational logic rules. Each of them combines a set of similar low-level transformations and describes exceptions that capture anomalies to a general change pattern.

Approaches discussed next are in the field of change analysis and classification. They are related to our taxonomy of source code changes.

Xing and Stroulia [XS05a] use their UMLDiff to classify interface changes. For each class version they assign a volatility level, *e.g.*, "intense evolution" or "rapidly developing," according to the number of changes occurred. In contrast, Kelter *et al.* focus on special differences of UML models (*e.g.*, attribute or reference differences), instead of general insert, delete, move, and update of UML diagram parts [KWN05]. Compared to their work, we classify individual changes.

Śliwerski *et al.* classify changes according to whether they induced a bug fix [SZZ05b], *i.e.*, changes that lead to problems. Their Eclipse plugin Hatari extracts and visualizes such changes [SZZ05a]. With our classification, we can detect frequent fix-inducing change types.

Small changes are also investigated by Purushothaman *et al.* in [PP05]. In a large case study, they found that there is less than a four percent probability that a one-line change will introduce a fault. This result implies that the significance

level of a one line change is low.

The area of code clone detection and software merging, although not directly related to our work, rely on source code differencing.

Sager *et al.* [SBPK06] used several tree matching algorithms for detecting similar Java classes. First, they convert the abstract syntax tree as generated by Eclipse into the language independent meta model FAMIX [DTS01]. In a second step, they transform the model into a generic tree format. The generic tree representations of all classes of a software system are then matched against each other to find similar classes. Sager *et al.* evaluated three different tree similarity algorithms for this purpose, derived from a *bottom-up maximum common subtree isomorphism*, a *top-down maximum common subtree isomorphism* and an *edit distance of two given trees*, all three originally presented in [Val02]. These algorithms can be used to replace the tree similarity measure calculated in our approach.

Baxter *et al.* describe *CloneDr*, a tool for code clone detection [BYdM+98] that relies on abstract syntax trees, but categorizes subtrees by hashing. This significantly reduces the number of comparisons needed, since only subtrees with the same hash values have to be compared. Classification using hash values works well for exact duplicates, but fails for locating near-miss clones, *i.e.*, code duplicates that are very similar. They are able to overcome this shortcoming by choosing an artificial bad hash function, *i.e.*, a function that ignores identifier names. For determining the similarity of two ASTs, Baxter *et al.* have used the Dice Coefficient [Dic45].

Mens has conducted a survey on existing software merging techniques in [Men02]. For example, the approaches presented in [Ask94, Wes91, Yan94, Hun01] rely on tree-based differencing in order to perform merging. All of them have some limitations concerning our concerns; as far as we know, neither of them detects *moves* or outputs an edit-script.

5.7 Conclusions

A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. Current approaches that investigate source code changes rely on information provided by versioning systems such as CVS. They track changes of source code files on a *text basis* without storing detailed information. In particular, the granularity, the type, and the significance level of changes between two versions of a source code entity are not tracked at all. To improve change analysis results, it is necessary to differentiate change types. Only in this way, can we provide better support for programmers, designers, and project managers to develop as well as maintain software systems and control their evolution.

To overcome the imprecise results of textual differencing, we presented *change distilling*, an approach for *fine-grained source code change extraction*. We enhanced the existing tree differencing algorithm of Chawathe *et al.* to classify source code changes according to our *taxonomy of source code changes* with the following substantial improvements:

- Using bi-grams as a robust string similarity measure that is able to cover common changes on source code identifiers.
- Adding a similarity check of node values to Chawathe *et al.*'s tree similarity measure to solve the problem of descendant subtree matching.
- Using inner node similarity weighting to reduce inadequate mismatches of condition expressions.
- Introducing the best match algorithm to reduce the impact of Chawathe *et al.*'s Assumption 1.
- Using dynamic thresholds to reduce the propagation of mismatches in small subtrees.

Furthermore, we introduced an extensive benchmark to evaluate source code change extraction algorithms. The benchmark consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. By applying the benchmark to the CHANGEDISTILLER, the implementation of our change distilling algorithm, we achieved significant improvements in extracting

change types: our algorithm approximates the minimum edit script by 45% better than the original change extraction approach by Chawathe *et al.* We were able to find all occurring changes and almost reach the minimum conforming edit script, *i.e.*, we reach a mean absolute percentage error of 34%, compared to 79% reached by the original algorithm.

Although our dynamic thresholds significantly inhibit mismatch propagation in small subtrees, we consider the problem as not fully solved yet. In our benchmark, we experienced such inadequacies with one particular method that is deeply nested and has major declaration changes. Since further improvements of string similarity measures are limited, we will investigate post-processing steps to filter further inadequate matches.

Acknowledgments

The work presented in this chapter was supported by the Swiss National Science Foundation as part of the COSE - Controlling Software Evolution project, and the Hasler Foundation as part of the ProMedServices - Proactive Software Service Improvement and EvoSpaces - Multi-dimensional Navigation Spaces for SW Evolution projects. The authors would like to thank Abraham Bernstein, Michele Lanza, Peter Vorburger, and the reviewers for their insightful suggestions that greatly helped to improve the original paper.

5.8 Additional Benchmark Results

For each of the four Runs (a)–(b) described in Section 5.5.3 the detailed results are listed in Tables 5.3 to 5.6. The tables contain the expected number of occurrences of each change type x, the found number of occurrences of each change type $\tilde{x}$, the mean absolute error ϵ, the root mean squared absolute error ϵ^2, the mean absolute percentage error δ, and the root mean squared absolute percentage error δ^2.

		(a)				
Change Type	x	$\tilde{x}$	ϵ	ϵ^2	δ	δ^2
Alternative Part Delete	9	32	1.04	1.24	0.08	0.29
Alternative Part Insert	15	40	0.86	1.13	0.06	0.22
Condition Expression Change	91	51	1.02	1.53	0.58	0.76
Method Renaming	1	1	0	0	0	0
Parameter Delete	9	12	0.43	0.65	0.07	0.19
Parameter Insert	16	20	0.29	0.53	0.04	0.13
Parameter Ordering Change	0	19	2.71	3.09	0	0
Parameter Renaming	3	1	0.67	0.82	0.67	0.82
Parameter Type Change	1	1	0	0	0	0
Return Type Change	1	1	0	0	0	0
Return Type Insert	1	1	0	0	0	0
Statement Delete	144	371	2.28	3.36	0.3	0.9
Statement Insert	391	640	2.15	3.41	0.4	1.08
Statement Ordering Change	14	105	2.26	3.52	0.12	0.49
Statement Parent Change	86	185	1.84	2.9	0.2	0.58
Statement Update	282	216	0.96	1.74	0.34	0.57
Total	1064	1696	3.27	6.44	0.79	1.66

Table 5.3: Additional Benchmark Results of Run (a)

		(b)				
Change Type	x	$\tilde{x}$	ϵ	ϵ^2	δ	δ^2
Alternative Part Delete	9	28	1.17	1.47	0.11	0.33
Alternative Part Insert	15	36	0.88	1.27	0.06	0.23
Condition Expression Change	91	64	0.89	1.43	0.44	0.68
Method Renaming	1	1	0	0	0	0
Parameter Delete	9	12	0.43	0.65	0.07	0.19
Parameter Insert	16	20	0.29	0.53	0.04	0.13
Parameter Ordering Change	0	19	2.71	3.09	0	0
Parameter Renaming	3	2	0.75	0.87	0.5	0.71
Parameter Type Change	1	0	1	1	1	1
Return Type Change	1	1	0	0	0	0
Return Type Insert	1	1	0	0	0	0
Statement Delete	144	283	1.96	2.72	0.29	0.93
Statement Insert	391	552	1.89	3.01	0.42	0.96
Statement Ordering Change	14	82	2.23	3.43	0.19	0.58
Statement Parent Change	86	194	1.87	3.05	0.21	0.63
Statement Update	282	318	0.72	1.32	0.19	0.51
Total	1064	1613	2.91	6.21	0.72	1.91

Table 5.4: Additional Benchmark Results of Run (b)

		(c)				
Change Type	x	$\tilde{x}$	ϵ	ϵ^2	δ	δ^2
Alternative Part Delete	9	25	1.06	1.33	0.12	0.34
Alternative Part Insert	15	33	0.78	1.14	0.07	0.23
Condition Expression Change	91	58	0.92	1.5	0.47	0.67
Method Renaming	1	1	0	0	0	0
Parameter Delete	9	11	0.33	0.58	0.08	0.2
Parameter Insert	16	19	0.23	0.48	0.04	0.14
Parameter Ordering Change	0	19	2.71	3.09	0	0
Parameter Renaming	3	1	0.67	0.82	0.67	0.82
Parameter Type Change	1	1	0	0	0	0
Return Type Change	1	1	0	0	0	0
Return Type Insert	1	1	0	0	0	0
Statement Delete	144	264	1.84	2.54	0.32	0.92
Statement Insert	391	533	1.76	2.84	0.38	0.89
Statement Ordering Change	14	83	2.08	3.34	0.11	0.38
Statement Parent Change	86	162	1.56	2.67	0.11	0.43
Statement Update	282	259	0.41	0.9	0.14	0.34
Total	1064	1471	2.2	4.89	0.52	1.45

Table 5.5: Additional Benchmark Results of Run (c)

		(d)				
Change Type	x	$\tilde{x}$	ϵ	ϵ^2	δ	δ^2
Alternative Part Delete	9	14	0.78	1.11	0.22	0.47
Alternative Part Insert	15	22	0.41	0.8	0	0
Condition Expression Change	91	85	0.58	1.01	0.24	0.48
Method Renaming	1	1	0	0	0	0
Parameter Delete	9	11	0.33	0.58	0.08	0.2
Parameter Insert	16	19	0.23	0.48	0.04	0.14
Parameter Ordering Change	0	17	3.4	3.61	0	0
Parameter Renaming	3	1	0.67	0.82	0.67	0.82
Parameter Type Change	1	1	0	0	0	0
Return Type Change	1	1	0	0	0	0
Return Type Insert	1	1	0	0	0	0
Statement Delete	144	225	1.77	2.37	0.34	0.79
Statement Insert	391	494	1.54	2.65	0.32	0.83
Statement Ordering Change	14	73	2.23	3.61	0.08	0.24
Statement Parent Change	86	118	1.14	1.64	0.21	0.53
Statement Update	282	260	0.41	0.9	0.14	0.34
Total	1064	1343	1.64	3.93	0.34	1.09

Table 5.6: Additional Benchmark Results of Run (d)

User Study

6

Evaluating a Query Framework for Software Evolution Data

Evaluating a Query Framework for Software Evolution Data. M. Würsch, E. Giger, H. C. Gall, ACM Transactions on Software Engineering and Methodology (TOSEM), Accepted for Publication, 2012

With the steady advances in tooling to support software engineering, mastering all the features of modern IDEs, version control systems, and project trackers is becoming increasingly difficult. Answering even the most common developer questions can be surprisingly tedious and difficult. In this paper we present a user study with 35 subjects to evaluate our quasi-natural language interface that provides access to various facets of the evolution of a software system but requires almost zero learning effort. Our approach is tightly woven into the Eclipse IDE and allows developers to answer questions related to source code, development history, or bug and issue management. The results of our evaluation show that our query interface can outperform classical software engineering tools in terms of correctness, while yielding significant time savings to its users and greatly advancing the state of the art in terms of usability and learnability.

6.1 Introduction

Over 90% of the costs during the evolution of software arise in its maintenance phase [Bro95]. One of the cost drivers is that many project managers staff their best people in the product team, while keeping the junior developers in the maintenance team. The latter are often overwhelmed, not only by the fact that they have to understand the code and design when they were not part of the team that made the decision, but also by their need to quickly gain proficiency in using the many software engineering tools that their development teams rely on. Integrated development environments (IDEs), version control systems, and project trackers—each of these systems provides a plethora of features of their own. Orchestrating them to answer questions, such as *"Who has recently changed this code and why?"*, is even more demanding and often involves tedious manual browsing through, for example, change logs and bug descriptions. Tool support that deals with the information needs of developers is therefore well-appreciated, and the integration of different software repositories is a hot topic in research and among tool vendors.

However, existing approaches that enable the integration of different information sources often do not allow developers to formulate ad-hoc queries. Instead, they need to be explicitly configured to enable new queries. On the other hand, query languages, such as CodeQuest [HVdM06] or JQuery [JV03], allow developers to formulate queries about software artifacts. These languages are usually based on an SQL- or Prolog-like syntax and effectively using them requires again considerable learning effort. According to Chowdhury, however, *"the most comfortable way for a user to express an information need is as a natural language statement."* [Cho04]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [Hen94].

We have therefore devised a framework that allows software engineers to use *guided-input natural language* strongly resembling plain English to query for information about the evolution of a software system. This includes queries related to source code, development history, as well as to bug and issue management. The framework builds on Semantic Web technologies, in particular ontologies, to formalize the knowledge that describes the data from these different domains. An early version of the framework was presented in [WGRG10] but was limited to

queries about static source code information only. Recently, we have extended our approach substantially and incorporated additional software evolution facets. In addition to source code, our framework now answers information needs related to the development history retrieved from version control systems, as well as bugs and issues reported in issue trackers. We have put further emphasis on data integration to enable queries that span multiple of these three facets. In this paper, we seek to answer the following three research questions:

- **RQ1:** How can we provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactical complexity?
- **RQ2:** When developers use such an interface to satisfy their information needs, are they able to successfully formulate and enter common developer questions, and can we observe an advancement over the state of the art in terms of time efficiency in retrieving the answers, as well as in the correctness of the answers?
- **RQ3:** Is the perceived usability higher for such an interface than for traditional means to access data about software systems, *i.e.*, those tools that are already provided by common IDEs, issue trackers, version control systems, and Web search engines?

At the heart of our paper is a user study, conducted with 35 subjects. Our study population, which provides a good approximation of junior developers, was given a set of 13 software evolution related tasks that we derived from common developer questions identified in the literature. The results of our study provide empirical evidence that subjects using our guided-input natural language interface achieve better performance in terms of correctness and time efficiency than with traditional software engineering tools. At the same time, the subjects are experiencing significantly higher system satisfaction.

The remainder of this paper is structured as follows: Section 6.2 gives a brief introduction to those concepts of the Semantic Web that the reader needs to understand in order to be able to follow the description of our approach. In Section 6.3, we present our framework to query software evolution knowledge with quasi-natural language. Its evaluation is discussed in detail in Section 6.4. Section 6.5 reviews existing work related to our approach and Section 6.6 concludes the paper.

6.2 The Semantic Web in a Nutshell

Berners-Lee *et al.* define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* [BLHL01]

Despite its origins, the Semantic Web is not limited to annotating webpages with meta-data. Virtually any piece of knowledge can be described in a computer-processable way by defining an ontology for the domain of discourse. According to Gruber, an ontology formally describes the concepts (classes) found in a particular domain, as well as the relationships between these concepts, and the attributes used to describe them [Gru93]. For example, in the domain of software evolution, we define concepts, such as *User*, *Developer*, *Bug*, or *Java Class*; relationships, such as *reports bug*, *resolves bug*, or *affects Java Class*; and attributes, such as *email address of developer*, *resolution date of bug*, *severity of bug*, etc.

The Resource Description Framework (RDF) [Ke04] is the data-model of the Semantic Web. The RDF data-model formalizes data based on *subject – predicate – object* triples, so called RDF statements. Such triples are used to make a statement about a resource of the real world. A resource can be almost anything: a project, a bug report, a person, a Web page, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [BLFM98]. In an RDF statement the subject is the URI of the thing (the resource) we intend to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model, information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [Pe08] can be used to query such RDF graphs.

RDF itself is domain-independent in that no assumptions about a particular domain of discourse are made. Specific ontologies have to be defined in an ontology definition language, such as the Web Ontology Language (OWL) [De04]. OWL enables the use of description logic (DL) expressions to describe the relationships between OWL classes. Instances of the latter are called *individuals* in OWL terminology and they belong to one or several classes. Class characteristics are specified by directed binary relations (predicates) called OWL properties. Object properties link individuals to individuals, whereas datatype properties link individuals to data values. Further constructs are provided for defining axioms about relation-

ships between properties (*e.g.*, inverse relationships), global cardinality constraints (*e.g.*, functional properties), and logical property characteristics (*e.g.*, symmetric or transitive properties). Properties may also have a domain and range. A domain axiom asserts that the subject of a property statement (a triple) must belong to the specified class. Similarly, a range axiom restricts the values of a property to individuals of the specified class or, in case of datatype properties, asserts that the value lies within the specified data range. There is also the notion of annotation properties (henceforth called simply *annotations*), which are comparable to comments in source code in their purpose. *OWL DL allows annotations on classes, properties, individuals, and ontology headers* [De04].

In addition to the W3C recommendations, the Semantic Web community developed tools to maintain and process RDF data. Jena[1] emerged from the *HP Labs Semantic Web Program* and recently became an Apache incubator project. It is a Java framework for building applications for the Semantic Web and provides a programmatic environment for RDF and OWL. Reasoners, such as Pellet [SPG+07], can be used in conjunction with Jena to infer logical consequences from a set of asserted facts or axioms. RDF databases, such as Apache Jena TDB, store RDF triples natively and can be queried directly with SPARQL.

[1] `http://incubator.apache.org/jena/`

6.3 A Quasi-Natural Language Interface for Software Evolution Data

In an earlier publication, we presented a framework for software engineers to answer common program comprehension questions with *guided-input natural language* queries [WGRG10]. Our approach supported queries about source code, such as those compiled by Sillito *et al.* [SMDV06]. Meanwhile, we have extended our framework to deal with questions related to various other facets of the evolution of a software system to support a wider range of common information needs of developers. This includes—besides source code—the development history, as well as the bugs and issues reported for large programs. The framework consists of a guided-input natural language interface, a set of ontologies that provide a formalization of software evolution knowledge, and a set of fact extractors to populate the ontologies with instances of real software systems. By *fact extractors* we mean parsers and algorithms that import software meta-data or *facts* from various software repositories, transform the extracted information into an ontology format, and store the results in a queryable knowledge base. Our approach is called HAWKSHAW[2] and Figure 6.1 gives an overview of its main components. Each box represents a component, whereas the arrows denote dependencies. Rounded corners are used for components that mainly serve as integrators for the data produced by the components they depend on. Boxes with dashed lines stand for third-party components. In the following sections, we explain each component briefly, starting with the HAWKSHAW query core that provides the algorithms that are used to guide developers in query composition.

6.3.1 Query Composition

The HAWKSHAW approach follows a method coined *Conceptual Authoring* or WYSIWYM (What You See Is What You Meant) by Hallet *et al.* [HSP07] and Power *et al.* [PSE98]. This means that, for composing queries, all editing operations are defined directly on an underlying logical representation, an ontology. However, the users do not need to know the underlying formalism because they are only exposed to a natural language representation of the ontology.

[2]We named our framework after Hawkshaw the Detective, a comic strip popular in the first half of the 20th century. *Hawkshaw* meant a detective in the slang of that time.

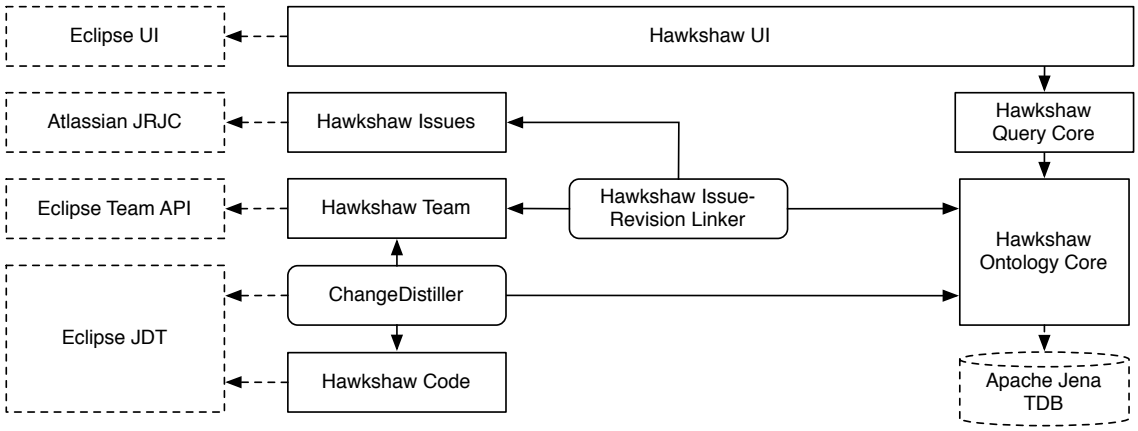


Figure 6.1: Hawkshaw Component Overview

Figure 6.2 shows a live example of the query dialog where a user has already started to compose a query: three words have been typed in so far, *"What Method invokes,"* and the drop-down menu presents the full list of concrete method names extracted from the source code that can be entered to complete the query. Figure 6.3 shows how the list of proposals changed over time before it reached its current state. The red letters mark the characters that were actually typed in by the user. The black ones were added by the auto-completion mechanism after a word was selected from the list of proposals. The list starts with the two words *"What "* and *"How."* Once *"What"* has been selected, the list will be rebuilt immediately with words that can follow the previous one. It is then again updated every time the user enters a character. In consequence, typing *"M"* will filter the list for words that start with that letter, such as *"Method"* and *"major."*

Users can type freely, as long as the entered characters match at least one of the proposed words. Therefore, our approach guides developers closely in formulating their information needs in a way such that the resulting query is processable by our query system. Our guided, quasi-natural language approach bears the following main advantages over free-form natural language queries: it is relatively light-weight in that it uses no linguistic processing at all (*i.e.*, no part-of-speech-tagging, stemming, etc.). Furthermore, developers—thanks to the proposals we show them—quickly receive feedback about the range of possible queries, and they are prevented from entering invalid questions not understandable by our query system. The immediate feedback helps to overcome the *habitability problem*

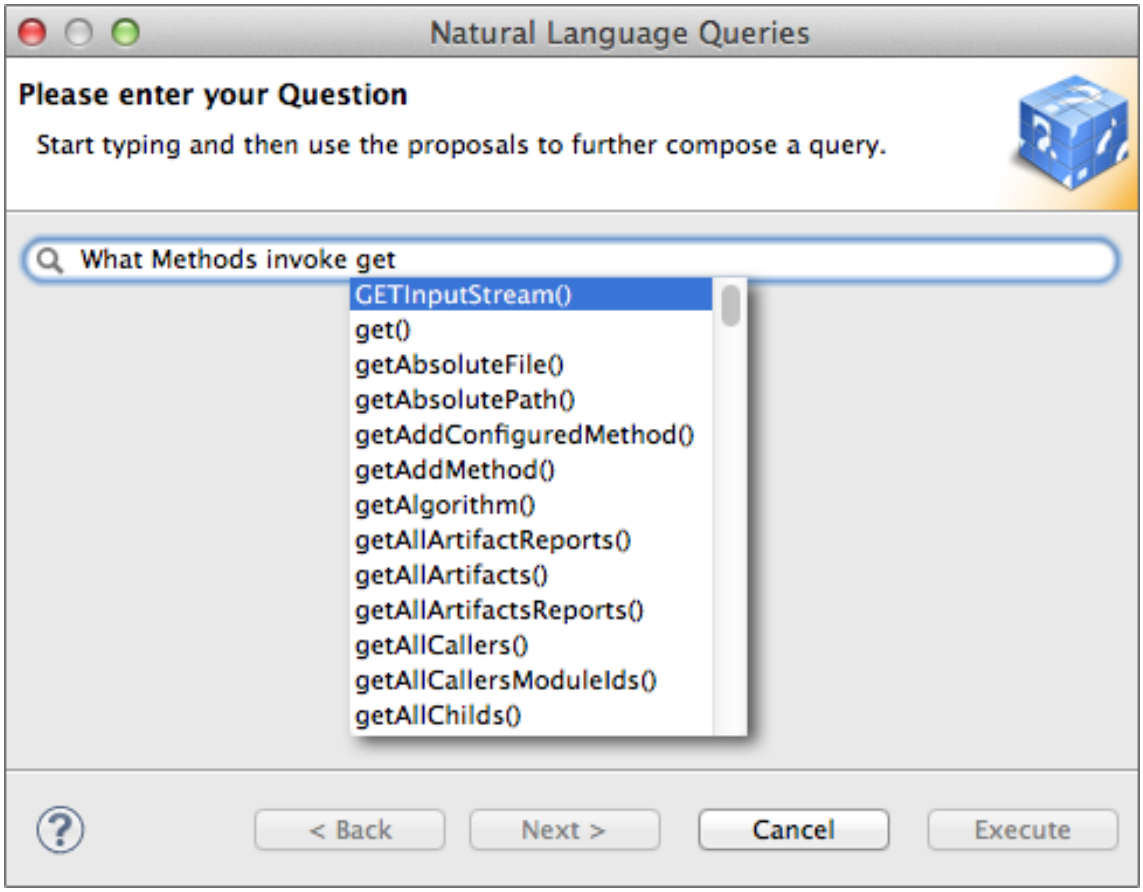


Figure 6.2: Screenshot of the Guided-Input Natural Language Interface in Action

formulated by Thompson *et al.* [TPT05], which occurs when there is a mismatch between the users' expectations and the capabilities of a natural language system.

Our implementation was originally based on the Ginseng tool by Bernstein *et al.* [BKKK06] but we have re-developed the interface and the underlying query composition system from scratch and integrated it seamlessly into the Eclipse IDE. The query dialog shown can be brought up anytime by pressing a shortcut. It is part of the HAWKSHAW UI component and also available from the *Search* menu of Eclipse. The integration goes as far as that users can directly reference editor selections of Java entities in their questions (*i.e.*, they can enter, for example, *"Where is **this** method called?"*, after having highlighted a method in the Java editor). The re-implementation was necessary to fulfill the scalability and flexibility requirements imposed by our user study. What remains from Ginseng is that we still use a multi-level grammar consisting of a static part that defines basic sentence structures and

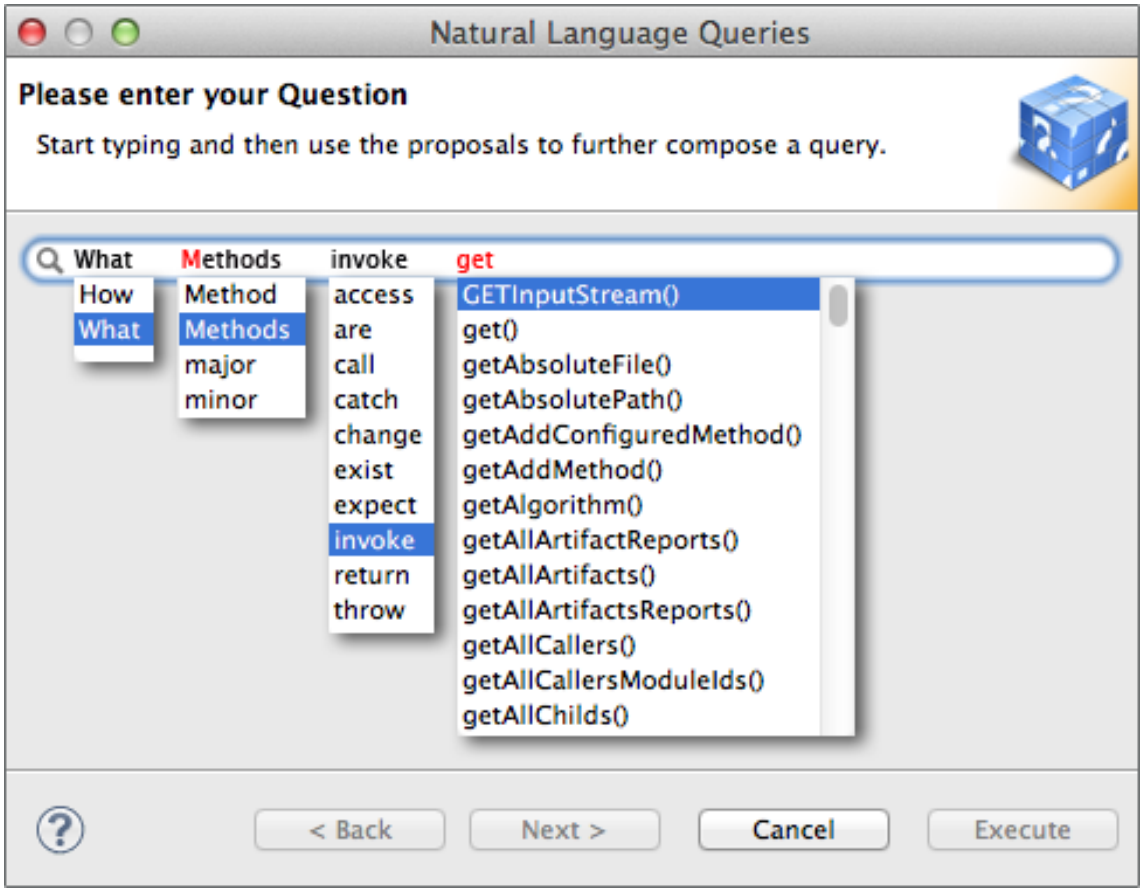


Figure 6.3: Screenshot of the Guided-Input Natural Language Interface illustrating how the List of Proposals changes over Time to reach the State shown in Figure 6.2

phrases for English questions, and a dynamic part generated at run-time from a knowledge base. The knowledge base is formalized with an ontology described in OWL and the data is serialized by means of RDF triples. The static part of the grammar needs to be defined manually and also contains information on how to compile the user input into the SPARQL query language.

The static grammar is part of the HAWKSHAW query core and consists of rules that basically act as a template for possible questions. The rules contain placeholders which are replaced on-the-fly during query composition by natural language labels and phrases extracted from the knowledge base. The labels and phrases constitute the dynamic part of the grammar, which is part of the HAWKSHAW ontology core. Table 6.1 lists nine out of over 80 rules taken from our grammar definition files. In practice, the rules follow a notation similar to

the *Extended Backus-Naur Form* but, for the sake of presentation, we introduce a simplified notation and omit the formal part needed for translation into SPARQL: Arrows denote that the right word or phrase follows after the left one, expressions in square brackets mark non-mandatory parts of a rule. Phrases in quotation marks represent statically encoded natural language parts. Each rule is terminated by a punctuation mark, in particular a period or question mark. Italic words are placeholders for dynamic rules generated from the ontology (*i.e.*, non-terminal symbols). The symbol $noun_c$ stands for nouns that are extracted from annotations of the OWL classes in our ontology (see Section 6.3.2 for more details on the annotations). Occurrences of $noun_i$ refer to labels of OWL individuals, *i.e.*, the instances of the OWL classes. The $verb$ and $adjective$ symbols are replaced by annotations of OWL object- and boolean datatype properties, respectively. Another case is $noun_{attr}$, which stands for annotations of non-boolean datatype properties.

Curly brackets can be used to mark the subject of a sentence, but are optional as long as a rule contains only one verb. The brackets can therefore be omitted in all but one of the example rules (Rule 5). The additional information is used together with the domain and range restrictions defined for the OWL object properties in our ontology to filter inappropriate words from the list of proposals. In other words, the verb of the sentence has to be an object property that fits the subject. That means that the object property needs to have the class in its domain that has been selected during query composition as the subject of the sentence. Object properties not fulfilling this constraint will not be presented to the user. Similarly, the object of the sentence must be an individual of a class in the ontology. The individual's class has to comply to the range of the object property, or it will not be shown either. For rules where the subject is not explicitly marked, we assume the last entered noun is the subject of the sentence.

Consider the following example, where a developer wants to find all methods that invoke another method with the identifier `bar()` and access a given field named `foo` (Rule 4 in Table 6.1). When the query dialog is brought up, a proposal list with words that are allowed to begin a question will pop up: *"Are"*, *"List"*, and *"What."* Then either a word can be selected from the list by clicking on it, or users can start to type in the letters. While typing, no longer relevant words are automatically filtered from the list. For example, if the word *"What"* is entered, Rules 2, 6, and 9 will no longer be relevant. The query system will then retrieve the next set of proposals according to the remaining six rules from the underlying

#	Rule	Example Question
	Open Questions	
1	"What" | "How many" → [$adjective$ →] $noun_c$ → "are there" | "exist" → ?	How many unresolved bugs are there?
2	"List" | "Give me" → "all" → [$adjective$ →] $noun_c$ → .	List all protected methods.
3	"What" | "How many" → [$adjective$ →] $noun_c$ → $verb$ → ["the" → $noun_c$ →] $noun_i$ → ?	What abstract classes implement the interface IFoo?
4	"What" | "How many" → [$adjective$ →] $noun_c$ → $verb$ [→ "the" → $noun_c$ →] $noun_i$ → [and → $noun_i$ →] ?	What critical bugs are blocked by #123 and #124?
5	"What" | "How many" → [$adjective$ →] {$noun_c$} → $verb$ [→ "the" → $noun_c$ →] $noun_i$ → [and → $verb$ → $noun_i$ →] ?	What methods invoke bar() and access foo?
	Closed questions	
6	"Are there any" → [$adjective$ →] $noun_c$ → "that" → $verb$ → $noun_i$ → ?	Are there any public methods that call bar()?
	Superlatives	
7	"What" → [$adjective$ →] $noun_c$ → $verb$ → "the" → "most" | "least" → $noun_c$ → ?	What class is affected by the most bugs?
	Comparisons	
8	"What" → [$adjective$ →] $noun_c$ → $verb$ → "more" | "less" → "than" → $number$ → $noun_c$ → ?	What method is called by more than 20 methods?
	Aggregations	
9	"List" → "the average" → $noun_{attr}$ → "of all" → $noun_c$ → .	List the average size of all methods.

Table 6.1: Static Grammar Rule Examples

knowledge base, in this particular example by retrieving all annotations of boolean datatype properties (*adjective*), as well as those of OWL classes ($noun_c$), since the *adjective* symbol is marked optional. Once the word *"method"* is selected, the query system will rebuild the proposals with *verbs* that are allowed to follow the previous noun, *i.e.*, with the annotations of those OWL object properties that have the OWL class `Method` in their domain restriction. One such object property is `invokesMethod`, which we have annotated in the ontology with verbs, such as *"invokes"* and *"calls."* Once selected, the range restriction of `invokesMethod` limits the next values for the $noun_i$ symbol to labels of those individuals that are instances of an appropriate OWL class. If the developer continues to compose the question like this, eventually the possibility will arise to either complete the

sentence with a question mark, or to continue by adding the conjunction *"and."* In the first case, the question will be compiled into a SPARQL query. The formal query is then automatically executed against the knowledge base and the results are displayed in a view similar to the one used by Eclipse for its Java Search results. In the second case, the query system will first retrieve annotations of the individuals represented by the $noun_i$ symbol that fit the last *verb* (Rule 4), as well as those object properties represented by *verb* that fit the $noun_c$ which was flagged as subject of the sentence (Rule 5). Then the query composition process continues until again the rule allows the question to terminate by the question mark.

In our previous work [WGRG10], we have shown that our framework already enables a wide range of queries comparable to that of other state-of-the-art research tools, such as the approach by De Alwis and Murphy [dAM08]. At the same time, HAWKSHAW exhibits more flexibility because it is not limited to a small set of concrete, hard-coded queries. For this paper, we have extended its querying capabilities even further, with additional knowledge extracted from version control systems or bug and issue trackers. The knowledge is used to generate additional dynamic rules, which—in combination with the static ones—allow for queries, such as *"What critical bugs affect this method?"* or *"Which developers changed this class?"* In the next section, we explain how we formalized the knowledge so that it can be queried with our approach.

6.3.2 SEON—Software Evolution Ontologies

Our approach makes heavy use of Semantic Web technologies. In the Semantic Web, knowledge is represented with ontologies, which in turn are described in terms of triples of subject, predicate, and object. This structure strongly resembles how humans talk about things and it can be easily transformed into natural language sentences. Specializations and generalizations allow to query knowledge on multiple levels of abstraction; for example, one can ask for *"person"* and also retrieve instances of developers and testers. Properties in OWL represent a binary relation that can be restricted by specifying domain and range. In triples this means that the domain restricts the possible values of the subject and the range restricts the values of the object. For our query approach, the explicit semantics can be exploited to filter the verbs that can follow a given subject, or the objects that can follow a given verb. All these features render ontologies a valuable knowledge representation format to be used in our approach.

The acronym SEON stands for *Software Evolution ONtologies*. It represents our attempt to formally describe knowledge from the domain of software evolution analysis and mining software repositories [WGH+12]. SEON covers a multitude of concepts but for the work presented in this paper only those related to source code, development history, issues and bugs, as well as fine-grained changes are of relevance.[3] In the following, we give a brief overview on these ontologies. For further details, we refer to the original paper about SEON.

Code Ontology. The source code ontology provides a formal meta-model for those source code entities and dependencies, which are common to many object-oriented programming languages. The formalization was done through the definition of OWL classes, object- and datatype properties. For example, we define OWL classes to represent packages, types, fields, methods and their parameters, etc. Object properties describe their relationships: classes are declared in a file and they declare members—methods and fields. The classes can inherit from other classes, methods invoke other methods, and so on.

Issue Ontology Modern project trackers often provide bug and issue management features. The data generated by those trackers is described by our issue ontology. The key concept is that of an issue, which can have several specializations, such as bugs, feature requests, improvements, etc. Both the key concept

[3]The full OWL definitions of SEON can be browsed online at: `http://se-on.org/`

and its specializations are represented by OWL classes. Issues have a description, usually written by their reporters. Eventually, they are assigned to developers for fixing them. Often, issues are triaged and classified by priority and severity. In consequence, we defined object properties, such as `fixesIssue`, `hasAssignee`, `hasPriority`, and `hasSeverity`, as well as the datatype properties `hasIssueNumber`, `hasDescription`, etc.

History Ontology. Data stored in version control repositories is represented by the ontology about the development history of software systems (*i.e.*, revisions or versions, releases, etc.). The central concept is that of a version of a file, which we have covered with the definition of an OWL class. Versions of files are committed by one specific developer and they have a commit message. A particular version of a file can be part of a release.

Change Ontology. Most version control systems are not aware of the exact syntax of the changes they maintain. Instead, they work on file-level and track only textual changes, *i.e.*, updates, additions, and deletions of lines. Our fine-grained change ontology describes changes made to source code down to the program statement level. For example, it allows us to describe the addition of a method invocation statement to a constructor of a class between two versions of a software system.

Natural language annotations provide human-readable labels for all classes and properties in SEON. They therefore bridge the gap between the machine-processable ontologies and the end-users of our quasi-natural language interface. For individuals, we use RDF Schema labels (rdfs:label) that are generated by our fact extractors. For example, the rdfs:labels of methods are simply their Java identifiers followed by parentheses, *e.g.*, *"println()."* The OWL class representing the concept related to Java methods has a custom OWL annotation property with the value *"method,"* whereas the object property `invokesMethod` describing a caller-callee relationship has the three annotations *"invokes,"* *"calls,"* and *"uses."* These words are used synonymously during query composition. HAWKSHAW extracts at runtime the natural-language annotations to guide developers in formulating questions, such as *"What method invokes...?"*, *"What method calls...?"*, and *"What method uses...?"*. By proposing a variety of synonyms, we take into account that developers use different nuances in terminology. With a reasonable selection of synonyms, HAWKSHAW can offer an experience that comes very close to free-form natural language input.

6.3.3 Fact Extraction

Next, we briefly describe how we populate our ontologies with instance data from real software systems under development. This happens with the aid of different fact extractors that first obtain data from various software repositories and then analyze different facets of the evolution of a software system. The fact extractors are implemented as a set of plug-ins for the Eclipse IDE and the extracted instance data is stored in a file-based Apache Jena TDB triple store. The integration of the different facets into one queryable software evolution knowledge base is described in Section 6.3.4.

Source Code Analysis. We use the Eclipse Java Development Tools (JDT) for the extraction of the static source code facts. In particular, we have implemented a custom project builder that maintains an up-to-date ontology model of the source code of a Java project stored in the local workspace of an Eclipse installation. The performance of the builder is comparable to that of the JDT builder during compilation of Java code. Parsing roughly 100k lines of code, including the creation and storage of the associated ontology model in TDB, takes less than one minute on a typical laptop computer.

Historical Analysis. To import and analyze the history of a project under version control, we interface the Eclipse Team API and retrieve, for each file, the full commit history. In consequence, with the HAWKSHAW tool, one is able to query the development history of any project no matter what version control system it uses—as long as there is a Team API compliant Eclipse plug-in available for that system. Analysis performance, however, strongly depends on the type of repository used. For example, the import of six years of development history with roughly six thousand different versions took about 20 minutes from a remote SVN repository. The analysis of the same number of versions, but from a GIT mirror of the same project, completed in under two minutes. The reason for the notable difference is that the SVN plug-in sends out many HTTP-requests, so that network performance becomes a limiting factor. For each GIT clone, in contrast, the whole change log is already available locally in compressed form.

Fine-Grained Change Extraction. During the historical analysis, we also run our fine-grained change extraction algorithm on each pair of consecutive versions [FWPG07]. CHANGEDISTILLER uses the Eclipse JDT to build an abstract syntax tree for each version and creates an edit script to transform the source code

of the older version into that of the newer one. The tree edit operations in the script are then classified into change types, such as *statement insert*, *method renaming*, and so on. The performance overhead of the change extraction is negligible on modern personal computers.

Issue Extraction. We have implemented a fact extractor for the Atlassian JIRA project tracker that is based on the JIRA REST Java Client (JRJC). The extraction performance depends on the network connection. For example, we imported over one thousand issues in under ten minutes from the Apache Foundation tracker.

6.3.4 Integration and Reasoning

Issue trackers and version control systems are information silos, with little to no integration between them and no possibility for performing cross-domain queries [Tap08]. To overcome this limitation of current software repositories, we perform additional integration steps to link issues to changes, as well as changes to code.

To find links to the issue database, we scan the commit messages of each version for references to bug and issue numbers. The scanning and linking can be done with a single, concise SPARQL construct query—at least for projects that define a rigid change process, with developers consistently referencing issues in each commit.

The corresponding SPARQL query is shown in Listing 6.1. When it is executed, the graph pattern consisting of the two triple patterns and a `FILTER`-expression in the `WHERE`-clause is matched against the triples of the RDF graph and returns the bindings for the variables in the `CONSTRUCT`-clause. SPARQL variables are indicated by the prefix "?" and, within a graph pattern, a variable must have the same value no matter where it is used. In the given query, the first pattern will match any triples where the predicate is the property `seon:hasCommitMessage`. Since the domain and range definitions of this property restrict the possible values of the subject and object, only revisions will be bound to `?v` and their commit messages to `?message`. Similarly, the second pattern will match against any statement with `seon:hasKey` as predicate and, consequently, the bindings for `?i` will contain issues and those for `?key` the corresponding issue keys generated by the issue tracker. The filter expression uses a regex function to narrow down the set of matching statements to those where the commit message of the

```
CONSTRUCT
  { ?v seon:fixesIssue ?i . }
WHERE
  { ?v seon:hasCommitMessage ?message .
    ?i seon:hasKey ?key .
    FILTER regex( ?message,
                  xpath:concat(?key, "[^0-9]"), "i") }
```

Listing 6.1: SPARQL Construct Query for linking Revisions to Issues

revision contains the key of the issue. For each pair of revisions and issues, the `CONSTRUCT`-clause will result in a new triple with the revision as subject, the property `seon:fixesIssue` as predicate, and the issue as object. We then add all resulting triples to our triple store.

Linking between versions and source code changes is done during fact extraction with CHANGEDISTILLER, where we explicitly state, for each extracted change, which source code entity was modified in what version. Now that issues are linked to versions, and the latter to code, we use the Pellet reasoner [SPG+07] to bridge the gap between source code changes and the issues that most likely caused them. For that, we defined rules, such as the following: When method m changes in version v, and version v is linked to issue i, then i affects m. The reasoner will apply that rule automatically to our ontology model and add the resulting triples to our triple store. Since we also use the reasoner to infer inverse properties (*e.g.*, `affectsIssue` is an inverse of `isAffectedByIssue`), we can then propose a multitude of domain-spanning questions to developers, for example *"What issues affected this method?"* or *"What classes were affected by issue #123?"*.

In summary, our approach combines industrial-strength technologies with ideas and tools from the Semantic Web to enable queries about software evolution artifacts in a way that comes natural to developers: using (quasi) natural language strongly resembling plain English. We use OWL to describe different software evolution artifacts and the relationships between them. A reasoner helps to make implicit knowledge explicit and therefore queryable. The resulting knowledge base then serves as input for our HAWKSHAW query interface. With the proof-of-concept implementation of HAWKSHAW,[4] we can answer our first research

[4]Hawkshaw is available for download at: `http://se-on.org/hawkshaw/`

question, RQ1: yes, with the components described in this section, it is possible to provide an integrated view on various facets of the evolution of a software system through an interface that exhibits the flexibility of formal query languages while avoiding their syntactic complexity. In the next section, we present an extensive user study to evaluate the remaining two research questions.

6.4 User Study

Our vision is to provide a convenient and intuitive interface that allows software engineers to access various kinds of knowledge related to the evolution of their software systems. To evaluate whether HAWKSHAW meets this claim, we designed and carried out a user study with 35 participants. In particular, we sought to answer the two remaining research questions with our evaluation:

- **RQ2:** When developers use such a quasi-natural language interface (*i.e.*, HAWKSHAW) to satisfy their information needs, are they able to successfully formulate and enter common developer questions, and can we observe an advancement over the state of the art in terms of time efficiency in retrieving the answers, as well as in the correctness of the answers?
- **RQ3:** Is the perceived usability higher for such an interface than for traditional means to access data about software systems, *i.e.*, those tools that are already provided by common IDEs, issue trackers, version control systems, and Web search engines?

We laid out the user study as a *Between Subjects Design* where the subjects were randomly assigned to either an experimental or a control group [WRH+00]. From the 35 subjects that participated in our study, 18 were assigned to the experimental and 17 to the control group. The experimental group was provided with HAWKSHAW. For the control group, we prepared a reasonable set of common developer tools. These tools served as a baseline to compare our approach against. The selection of the baseline is discussed thoroughly in Section 6.4.4.

We then assigned the same set of 13 software evolution tasks to both groups and defined three hypotheses based on RQ2 and RQ3 to statistically validate the outcome of our study. The software evolution tasks are introduced in Section 6.4.1 and our hypotheses are listed in Table 6.2.

ID	Null Hypotheses
$H1_0$	The distribution of the *Total Score for all Tasks* is the same across the experimental and control group.
$H2_0$	The distribution of the *Total Number of Seconds spent for solving all Tasks* is the same across the experimental and control group.
$H3_0$	The distribution of the *System Usability Scores* is the same across the experimental and control group.

Table 6.2: Hypotheses

6.4.1 Choosing the Tasks

Finding a representative set of tasks is fundamental for the validity and generalizability of a user study. We therefore surveyed the literature for previous experiments and existing catalogues of common developer questions in the context of program comprehension or software maintenance and evolution. Our aim was to find relevant questions related to source code, development history, and issues, which would allow us to directly compare HAWKSHAW with tools that are widely used in industry. We further looked for questions spanning multiple facets to evaluate our integrated approach.

We compiled our set of 13 tasks from the work of LaToza *et al.* [LVD06], Sillito *et al.* [SMDV06, SMV08], Ko *et al.* [KDV07], De Alwis and Murphy [dAM08], Fritz and Murphy [FM10], and Hattori *et al.* [HDLL11]. These tasks, along with brief explanations for why we claim they are relevant, are listed in Table 6.3. We anonymized developer names, issue numbers, Java identifiers, etc., for publication but the full questionnaire can be obtained from the corresponding author of this paper. With the rationale listed in the table, we describe the importance of each task in our own words.

We intended to stay as close as possible to the original text of the questions from the literature but found it necessary to adapt them with light modifications: the questions were made more specific (where necessary) in order to reduce the range of possible interpretations. For example, the high-level question *"What have my coworkers been doing?"* [KDV07] was reformulated into *"What feature requests were implemented by Developer$_2$?"*. The original question could also be interpreted in numerous other ways, *e.g.*, *"What classes or files were committed by Developer$_X$?"*, so we removed the potential source of confusion for the study subjects to facilitate scoring of correct answers.

ID	Tasks
	Code Domain
T1	**Description.** All the subclasses of $Class_1$? [SMV08] **Rationale.** When a base class is modified, its subclasses will be affected. Finding those classes quickly is the first step in assessing the change impact. Also helpful for program comprehension, *e.g.*, when searching for implementations of an abstract class.
T2	**Description.** All methods with $Class_2$ as argument (parameter)? [SMV08] **Rationale.** Identifying methods that can operate on an instance of a given class can reveal useful API that supports the task at hand.
T3	**Description.** All methods that invoke both, $Method_1$ and $Method_2$? [SMV08] **Rationale.** Finding pieces of code where a method is referenced is crucial for program understanding and also for finding proper API usage examples. Often, it is also helpful to combine the result from two distinct searches, *e.g.*, when checking for violations of common idioms.
	History Domain
T4	**Description.** The developers who have changed $Class_3$ in the past? [HDLL11] **Rationale.** Team activity awareness is helpful for coordination and for finding experts of a particular part of a software system.
T5	**Description.** The file that has changed most often in the past? [FM10] **Rationale.** An excessive amount of changes can indicate a design problem and reveal candidates for refactoring in order to improve the separation of concerns.
T6	**Description.** The last five files changed by $Developer_1$? [HDLL11] **Rationale.** When one has to replace a previous project team member, it is important to quickly get an overview on the member's previous work. One way to achieve this, is by looking at the code the team member was working on.
	Issue Domain
T7	**Description.** What feature requests were implemented by $Developer_2$? [KDV07] **Rationale.** Similar to Task 6, the goal is to become familiar with someone else's work. This time from a different angle, *i.e.*, by looking at the kind of features the developer was responsible for.
T8	**Description.** The issues $Developer_3$ and $Developer_4$ commented on? [FM10] **Rationale.** When relying on third-party libraries, developers often comment on bug reports in those libraries that affect their work. Retrieving these bugs later is useful for various reasons, such as team awareness, time tracking, and checking if workarounds are no longer necessary.
T9	**Description.** The issues blocked by $Issue_1$? [common feature of issue trackers] **Rationale.** As soon as an issue is resolved, other issues previously blocked by the current one can move into the center of attention.
	Cross-Domain
T10	**Description.** The classes affected by $Issue_2$? [LVD06] **Rationale.** Quickly assessing the impact of an issue is useful for effort measurement.
T11	**Description.** The issues that affected $Class_4$? [LVD06] **Rationale.** Understanding change history if a piece of code is the first step in understanding the design decisions behind its implementation. Previous issues affecting the code explain some of the reasons for change.
T12	**Description.** The most error-prone class? [KZWJZ07] **Rationale.** Kim *et al.* have reported *that faults do not occur in isolation, but rather in bursts of several related faults.* In consequence, past bugs are a good predictor for future bugs. Finding classes affected by many bugs therefore can help allocating resources for testing efficiently.
T13	**Description.** The issues that affected $Class_5$ and $Class_6$? [LVD06] **Rationale.** Finding issues affecting different classes can reveal the reasons behind logical coupling.

Table 6.3: Task Description and Rationale

Because of HAWKSHAW's conception as a quasi-natural language interface, developers can enter many other questions found in the literature without further transformations. While this is an important feature of our approach, we still tried to re-formulate the questions to neutral sentences so that we did not treat our approach preferentially. In a few cases, we added twists to evaluate specific aspects. *"What calls this method?"* [dAM08], which can be entered directly in HAWKSHAW, became Task 3: *"All methods that invoke both, Method$_1$ and Method$_2$."*

In general, we paid close attention not to penalize the baseline tools in comparison to our approach. For example in Task 3, where the intersection of the callers of two different methods A and B is requested, we selected methods with a low number of callers to facilitate the composition of the partial results for the control group. One of the methods had 15 different callers, but the other one had only two—coming up with the common callers, hence, was trivial. Similarly for Task 6: *"The last five files changed by Developer$_1$?"*, where we looked for a developer that had recently committed five files, so that the participants of the control group did not have to browse through many revisions. In fact, the result was to be found already among the five top-most rows of the *History View* of Eclipse.

We decided to exclude questions concerning the fine-grained change history provided by CHANGEDISTILLER from the user study. The decision was made after we had completed a pre-study, which showed us that such tasks—especially in comparison with our approach—are poorly supported by Eclipse and therefore hardly solvable within the given time. Questions, such as "What developers changed Method$_X$?", can be answered with HAWKSHAW right-away but often involve laborious differencing with the *Compare*-feature in Eclipse.

In summary, we are convinced that we selected tasks that do not unduly favor our HAWKSHAW tool and that these tasks relate to valid, common information needs of software developers. To further support this claim, we additionally asked the participants of our user study to rate each task with respect to its degree of realism, *i.e.*, whether they would be likely to solve a task similar to the one at hand in practice. These ratings are presented in Section 6.4.9, however they have to be considered with care because our subjects were mostly students with limited industrial experience.

6.4.2 Evaluating Usability

We used standardized satisfaction measures to investigate how user-friendly our study subjects perceived the HAWKSHAW approach. The same measures were applied to the baseline tools for comparison. After a thorough evaluation of usability-related measures, we decided for the *System Usability Scale (SUS)* by Brooke [Bro96], a popular questionnaire for end-of-test subjective assessments of usability. The SUS a de-facto industry standard, used in hundreds of publications, and its robustness and reliability have been confirmed empirically by, amongst others, Bangor *et al.* [BKM08].

An advantage of the SUS over alternatives, such as the *Post-Study System Usability Questionnaire (PSSUQ)* [Lew92], is its conciseness; it can be filled out quickly by the subjects, lowering the risk of incomplete or non-serious responses. The SUS consists of the following ten items, each with five response options that range from "Strongly Disagree" to "Strongly Agree":

1.) I think that I would like to use this system frequently.
2.) I found the system unnecessarily complex.
3.) I thought the system was easy to use.
4.) I think that I would need the support of a technical person to be able to use this system.
5.) I found the various functions in this system were well integrated.
6.) I thought there was too much inconsistency in this system.
7.) I would imagine that most people would learn to use this system very quickly.
8.) I found the system very cumbersome to use.
9.) I felt very confident using the system.
10.) I needed to learn a lot of things before I could get going with this system.

We asked our subjects, after they had completed all the 13 tasks, to record their immediate response to each item (as recommended by the author of the SUS), rather than thinking about items for a long time. The experimental group had to answer with respect to their experience with HAWKSHAW, whereas the control group had to reflect on the features of the baseline toolset that they actually used for solving the tasks. We then aggregated the responses into a single number for each subject, representing a composite measure of the overall usability of

HAWKSHAW and the baseline, respectively. The exact scoring is described in Section 6.4.7.

In addition to the measurements provided by the post-test questionnaire SUS, we measured user satisfaction immediately after the completion of each task. Among the numerous questionnaires available to gather post-task responses, we selected the *Single Ease Question (SEQ)*. According to Sauro and Dumas [SD09], the SEQ exhibits the important psychometric properties of being reliable, sensitive, and valid—while also being short, easy to respond, and easy to score. It consists of the single question *"Overall, this task was?"* in combination with a seven-point Likert-scale that ranges from *"Very Difficult"* to *"Very Easy"*. Sauro claims that:

> *"The beauty of the SEQ is that users build their expectations into their response. So while adding or removing a step from a task scenario would affect times, users adjust their expectations based on the number of steps and respond to the task difficulty accordingly."* [Sau10]

This property is important, because it allows us to differentiate between the cases where a task was actually difficult to solve with the available tools, and those when obtaining the correct solution was simply laborious, but less mentally challenging.

6.4.3 Research Population

Our research population included 35 subjects, of whom 25 were advanced undergraduate students and eight graduate students. The remaining two participants were post-doctoral researchers. We recruited the subjects among the approx. 80 participants of two different courses: a fourth-semester lab course on software engineering and an advanced course on software evolution and maintenance. Since an internship in industry is part of the curriculum, the students of the advanced course already had industry-level software development experience. The experimenters were involved in neither of the two courses and the participation in our study took place on a completely voluntary basis for our students. However, we rewarded them with a small monetary compensation for their time. We clearly communicated that we were evaluating exclusively our approach—not the participants—and that we respected the anonymity of the subjects at all times.

The average subject's age was 25.2 years, ranging from a minimum of 20 to a maximum of 39 years. In Figure 6.4, an overview of the number of years of

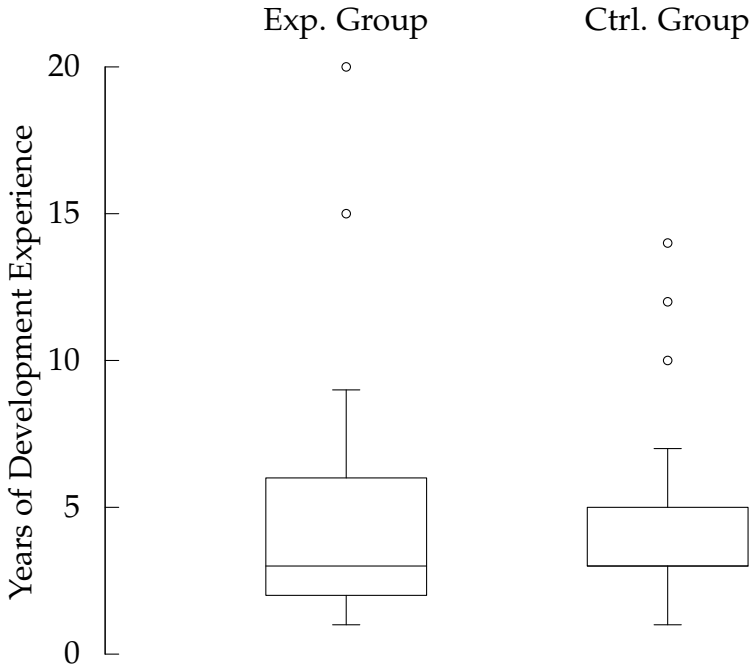


Figure 6.4: Number of Years of General Development Experience of the Experimental Group (18 Subjects) and Control Group (17 Subjects)

development experience in general (*i.e.*, no matter what programming language and IDE) is given for the subjects of both groups. Figure 6.5 shows how the Java-specific development experience was distributed for both groups. Overall, the average development experience of the participants was five years with a median of three years, ranging from the minimum of one to a maximum of 20 years. Their particular development experience with Java and the Eclipse IDE was between one and six years, with 2.8 years on average and a median of two years.

We further asked the participants to do a quick self-assessment of their skills in English, coding in general, Java in particular, JIRA, and SVN. An overview on the self-assessment of the experimental and control group can be found in Figure 6.6 and Figure 6.7, respectively.

In summary, it shows that our research population is a good approximation to junior developers, which we expect to be the user group that benefits most from our HAWKSHAW approach.

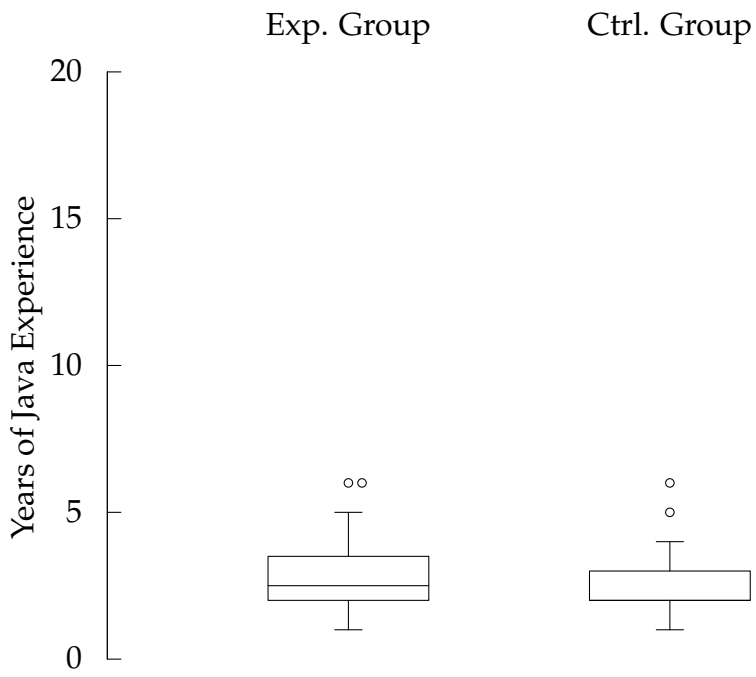


Figure 6.5: Number of Years of Java-specific Development Experience of the Experimental Group and Control Group

6.4.4 Finding a representative Baseline Toolset for the Control Group

The tasks of our user study involved questions about source code, the development history, and issues. While our HAWKSHAW approach integrates this wide array of information in one queryable ontology model, most modern IDEs provide only a subset of the features out of the box that are necessary to access the same information directly from within the IDE. We therefore prepared a small set of commonly used developer tools as a baseline for comparison with our approach.

The baseline tools were comprised of all the features of the Eclipse Classic package (v3.7.2), the Subversive SVN plug-in for Eclipse, and a Web browser for conducting Web searches and accessing the Web front-end of the Atlassian JIRA project tracker. Eclipse provided source code browsing and search features and was already well-known to most participants of our study. The majority of our participants were already familiar with SVN and the Subversive Eclipse plug-in, since they had actively used it in one of their lab courses for a few months in the

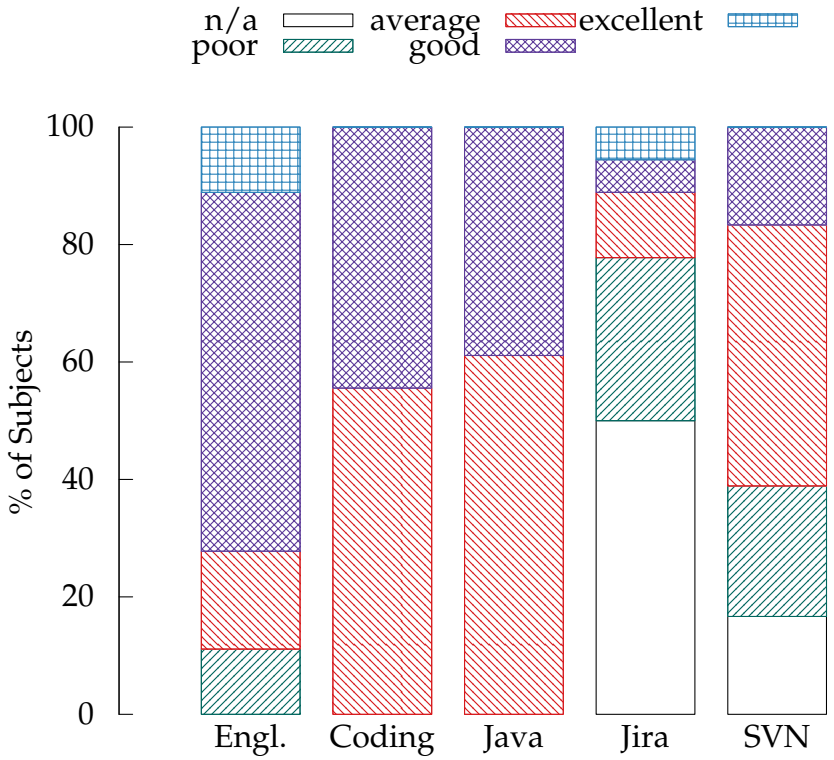


Figure 6.6: Skill Self-Assessment of the Experimental Group (18 Subjects)

context of a team development project. The Subversive plug-in contributes the *History View* to Eclipse that allows browsing the change log of a project or of single files.

We have chosen to study an open source project, which uses JIRA for its issue management (see Section 6.4.5 for details on the selected project). Besides the popularity of JIRA, we found the feature set provided by its Web front-end to be well-suited for direct comparison with our HAWKSHAW approach. Most importantly, JIRA provides tight integration with SVN: for each issue, the related SVN commits with the involved files are listed in the Web interface. The SVN integration compensates for the lack of traceability between issues and changes in standard Eclipse installations.

All these tools provide a strong baseline with a variety of features that left the control subjects well-equipped to solve the tasks listed in Section 6.4.1. We also claim that the baseline is representative for the tools that many developers employ nowadays in practice.

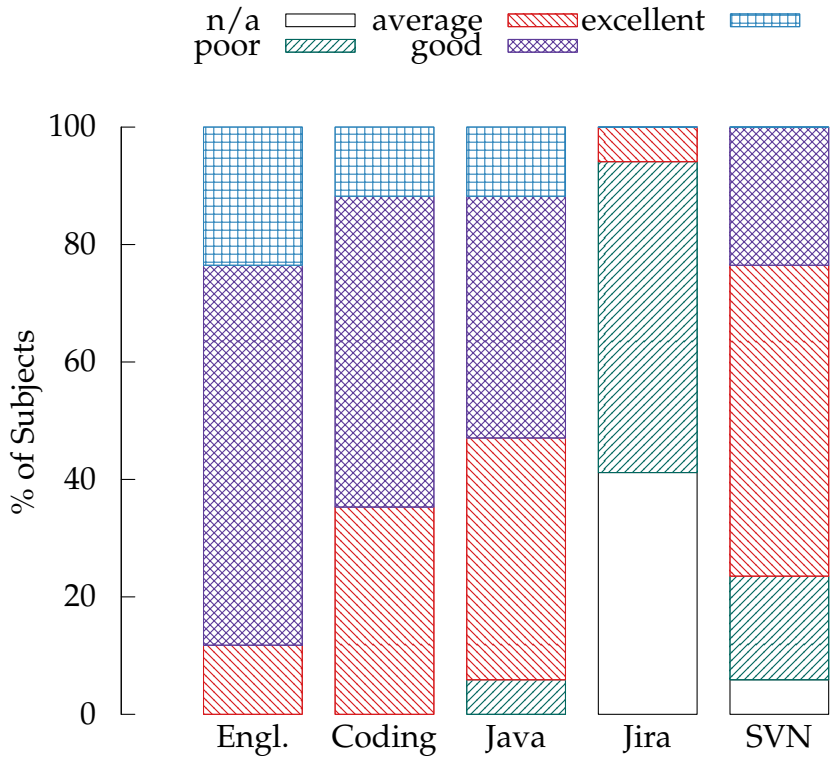


Figure 6.7: Skill Self-Assessment of the Control Group (17 Subjects)

6.4.5 Choosing a Study Object

The tasks presented in Section 6.4.1 were carried out by the participants on a real software system: we selected the Apache Ivy project,[5] consisting of 103,647 source lines of code (SLOC) in 919 Java classes at the time of the user study. It is a popular open-source dependency manager and has been used as a case study in an increasing number of publications recently, for example, in [PFD11, BAJ11]. Ivy has been under development for about six years under the patronage of the Apache Foundation, where a team of seven developers has contributed to it. We have analyzed and imported with HAWKSHAW more than six thousand revisions of Java files from the SVN repository of Ivy and more than one thousand issues from its JIRA tracker.

Our selection was based on the consideration that the subjects should answer questions about a real-world, industry-scale project to increase the external validity

[5] `http://ant.apache.org/ivy/`

of our study. However, only such systems were considered by us that were written in Java and neither exceeded a reasonable size, nor belonged to a very complex application domain. These restrictions were imposed by the short amount of time the participants had to familiarize themselves with the system, as well as by the average skill and experience level of our participants.

We further looked for projects that use SVN for version control and JIRA for issue tracking, and that implement a rigid change process. For such projects, our fact extractors can reliably link JIRA issues to SVN revisions. Both criteria are met by Ivy: its developers use SVN and each commit message contains a reference to the corresponding change requests (*i.e.*, the issue numbers).

6.4.6 Conducting the User Study

We carried out the user study of HAWKSHAW during three sessions. Each session lasted for approx. 45 to 80 minutes. The study took place in the computer labs of the Department of Informatics at the University of Zurich, where we had installed HAWKSHAW and the baseline tools in advance.

Each session started off with a short introduction, where we outlined the goals of the study. We quickly went through each page of the questionnaire to make sure that the members of the experimental and control group understood what we were asking of them. The subjects were explicitly told what tools they could use and which they were not allowed to rely upon. The complete instructions were replicated on the front-page of the questionnaire, which existed in two variants: one for the experimental- and one for the control group. The tasks for both groups were identical but we gave the control group some hints on which of the baseline tools could help in solving each subset of tasks. For example, for the tasks involving questions about issues, we mentioned that the JIRA tracker of the Apache Ivy project could yield interesting insights and listed its Web address. Or, as another example, the *History View* of Eclipse was suggested for solving the tasks related to the development history.

We prepared a short tutorial for each group in the form of a PDF document which we provided to the participants at the beginning of each session. It contained annotated screenshots of the tools, as well as some brief explanations of the tools' features needed for succeeding in the study. The tutorial for the experimental group explained the HAWKSHAW query interface. Additionally, it contained a few

general tips for effective querying, *e.g.*, that the subjects should only formulate questions with subject-predicate-object structure and in present tense. We gave them only two positive and one negative examples for queries with HAWKSHAW—all three of them unrelated to the tasks in the study. In the control group's tutorial for using the baseline tools, we explained the source code search features of Eclipse. This included a brief user guide on the *Type Hierarchy View*, the *Call Hierarchy View*, the *Find-References* command, as well as on the *Show-History* command and the associated *History View*. We further demonstrated the use of the JIRA Web front-end by labeling all important widgets on screenshots and explaining how the simple and the advanced search works. The detailed tutorial in conjunction with their previous Eclipse and SVN experience left the members of the control group in a highly-competitive starting position compared to their counterparts in the experimental group, which could only rely on our tutorial but never had used HAWKSHAW before.

We did not restrict the time for reading the tutorial and it was up to the subjects to start working on the tasks when they felt ready for it. However, once they began solving the first task, we imposed strict time limits on them: for each task, the subjects were given five minutes. Once this amount of time passed, the participants had to write down their answer and then proceed to the next task.

6.4.7 Data Collection and Pre-Processing

Throughout the course of the study, we collected various data, which we then pre-processed and analyzed to obtain the empirical results described in Section 6.4.8. The collection and pre-processing steps are outlined below.

Personal Data. Prior to the study, we asked the participants to share some personal information with us, such as age, gender, current occupation, and their English skills. We also enquired about their expertise with software development in general, as well as about their Java and Eclipse skill in particular. Finally, they were also asked about their level of familiarity with SVN and JIRA.

Correctness Data. To map each subject's solutions to scores comparable with those of others, we used the following simple scoring scheme: each of the tasks, when solved correctly, was rewarded with one point. So a maximum score of 13 points could be achieved. In case of incomplete solutions, each correct item was worth one point divided by the number of total correct items. For example, the correct solution to Task 7 consists of two Java classes. In case that a subject wrote down only one answer out of two, we scored the solution with 0.5 points. We decided not to penalize incorrect answers; in the previous example, if the subject wrote down a third unrelated class, the total score for that task was not diminished. The model solution (or oracle) was defined by the authors with aid of the baseline tools. Then it was carefully validated with HAWKSHAW to ensure that no false positives or true negatives remained. In the remainder of the paper, we will use the term *correctness* whenever we talk about these scores; if, for example, we say that group x achieved an average correctness of 6.5, we mean that the participants of the group received on average 6.5 out of 13 possible points.

Timing Data. We asked the participants to solve the tasks as quickly as possible and in the given order. To time the subjects accurately, we contributed an *Experiment Timer View* to Eclipse. The view showed the current task and a progress bar with the remaining time for solving that task. The subjects had to start the timer themselves, once they had read the tutorial and provided their personal information. In case the timer expired, a popup and an audible notification urged the participants to write down their (partial) answer and proceed to the next task. When the participants finished a task early, they could click a button to proceed. In any case, the application displayed a reminder to write down the elapsed time on the questionnaire before restarting the timer for the next task.

Usability Data. To measure usability of HAWKSHAW in comparison to the baseline tools, we incorporated the SUS questionnaire (*cf.* Section 6.4.2). The SUS yields a single number between zero and 100, representing a composite measure of the overall usability of the system being studied. The score is based on the individual answers for each of the ten items of the SUS. We then applied the original SUS scoring scheme:

> *To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4. For items 1, 3, 5, 7, and 9 the score contribution is the scale position minus 1. For items 2, 4, 6, 8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by 2.5 to obtain the overall value of SU.* [Bro96]

Besides the classical SUS score, which represents a global measure of system satisfaction, recent research has discovered empirical evidence for two sub-scales of usability and learnability [LS09]. In particular, the fourth and tenth items provide the learnability dimension and the other eight items provide the usability dimension. For both, the contributions per item are calculated based on the scoring scheme above. However, the sum of the items related to learnability and the sum of those related to usability is then multiplied by 12.5 and 3.125, respectively.

Qualitative Feedback. In addition to the quantitative data gathered, we asked the subjects for their opinion on our quasi-natural language approach, as well as on the user study itself. In particular, we asked for any comments and/or suggestions that could improve the user study. The members of the experimental group additionally could comment on what they liked the most/least of HAWKSHAW. They were also given the opportunity to list any questions that they wanted to enter with our approach but did not succeed in doing so. The control group was asked for features of Eclipse and JIRA that were most helpful to them during the study, and whether they were missing any functionality or found something particularly difficult to use.

6.4.8 Empirical Results: Overview

We first present the statistical analysis of the aggregated results of our user study. In Section 6.4.9 we then describe the data we obtained for each of the individual tasks. The results of the study are summarized in Section 6.4.12 and we discuss potential threats to validity that may have arisen from our study design in Section 6.4.11. The interpretation and discussion of the empirical results is given in Section 6.4.12.

To support our choice of an appropriate statistical test for our hypotheses, we analyzed the distributions of the overall results on correctness, completion time, and usability scores—as well as the data for each individual task. Although the *Shapiro-Wilk Test of Normality* hinted at normally distributed data, a visual analysis of the corresponding histograms and the Q-Q plots raised reasonable doubts on normal distribution—especially when looking at how the individual results for each task were distributed. We performed global tests to accept or reject the hypotheses given in Table 6.2. In addition to the global tests, we performed post-hoc tests for the individual tasks to break down the overall results into detailed results for every task. We therefore decided to use the non-parametric Independent Samples Mann-Whitney U (MWU) test with a significance level $\alpha = 0.01$. The post-hoc tests are presented in Section 6.4.9. Their purpose is to identify how each task contributes to the overall results.

Overall Correctness. The subjects in the experimental group achieved on average a 59.13% higher correctness than those in the control group: the mean total correctness for the experimental group is 10.20 out of 13 possible points with a standard deviation (std dev.) of 2.00 and a median of 10.50. For the control group, we observed a mean of 6.41 with a std dev. of 2.27 and a median of 6.00. The box plots in Figure 6.8 further illustrate that the 25^{th} percentile of the experimental group lies above the 75^{th} percentile of the control group, which means that 75% of the subjects in the experimental group reached a higher total correctness than 75% of their counterparts in the control group.

We tested the two distributions for equality. The MWU test showed that their difference is significant at the 99% confidence level (p-value=2.66E-5). As a consequence, we reject $H1_0$ (*cf.* Table 6.2) and accept the alternative hypothesis that the distribution of the total number of correct answers is different across the two groups.

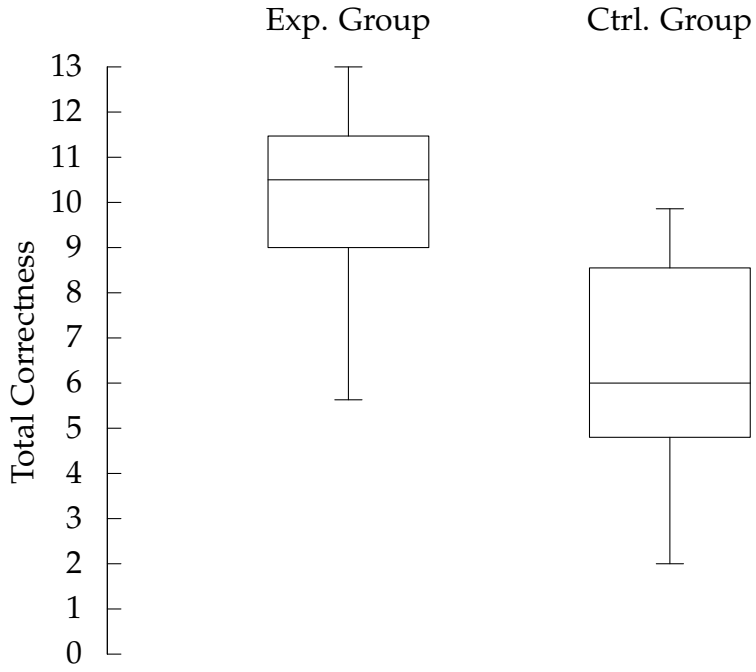


Figure 6.8: Total Score per Group

Overall Completion Time. With our approach, participants were 35.41% faster than the baseline. This is equivalent to average time savings of 16 minutes and 54 seconds for all the 13 tasks. The experimental group needed, on average, 30 min 50 sec to solve all tasks—with a std dev. of 6 min 61 sec and a median of 31 min 37 sec. The mean for the control group is 47 min 44 sec, the std dev. is 7 min 4 sec, and the median is 46 min 54 sec.

In Figure 6.9, we depict similar results with respect to completion time, as we had previously made for correctness: The 75^{th} percentile of the box plot for the experimental group is below the 25^{th} of the box plot for the control group. In other words, three quarters of all the subjects with HAWKSHAW could solve the tasks faster than three quarters of the subjects with the baseline tools.

The MWU test rejects the null-hypothesis $H2_0$ at the 99% confidence level with a p-value of 6.13E-8. The acceptance of the alternative hypothesis confirms that the distribution of the total completion times among the two groups is different. With the acceptance of $H1$ and $H2$, we can now answer RQ2: developers can indeed successfully formulate and enter common developer questions with our quasi-natural language interface. Overall, HAWKSHAW provides a clear advancement

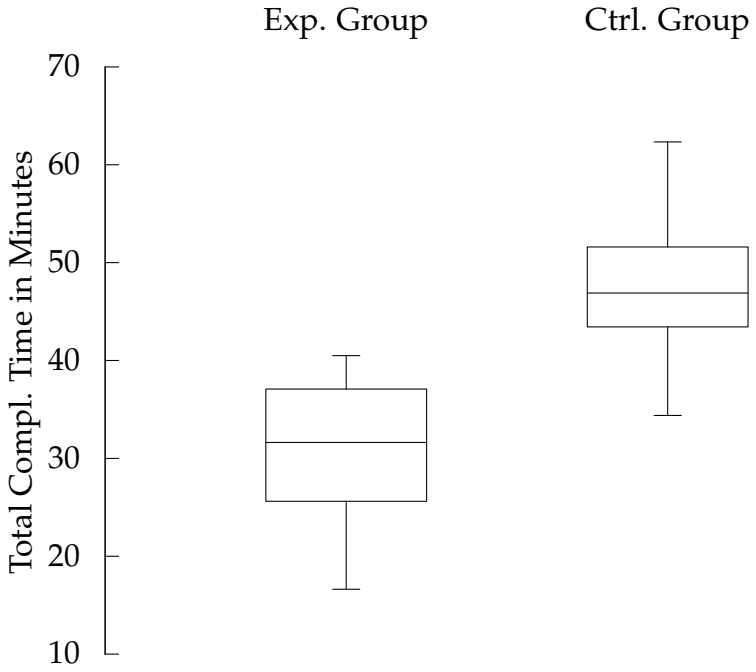


Figure 6.9: Total Completion Time per Group

over the baseline tools with respect to both correctness and time efficiency.

Overall Usability. The mean of the SUS score for HAWKSHAW is 42.02 points higher than the mean of the control group's score, that is 76.11 versus 33.09 out of a theoretical maximum score of 100. The std dev. for the first group is 14.46, and for the second one 14.81. This translates to high sub-scores for HAWKSHAW in usability (mean=74.31, std dev.=15.18, and median=73.43) and learnability (mean=83.33, std dev.=16,61, and median=87.5), whereas the baseline tools were rated significantly lower in both usability (mean=31.8, std dev.=14,91, and median=34.38) and learnability (mean=38.24, std dev.=21.86, and median=37.5).

The box plots in Figure 6.10 show that the 25^{th} percentile of the experimental group is again above the 75^{th} percentile of the control group, denoting that HAWKSHAW was well-accepted by the majority of its users, whereas the subjects of the control group were, overall, less satisfied with the baseline tools.

Sauro provides guidance in interpreting SUS scores based on a comprehensive study [Sau11]. In analyzing data from over 5000 users across 500 different evaluations, the author determined an average score of 68 for the systems tested and also calculated the percentile ranks for different ranges of scores. The SUS

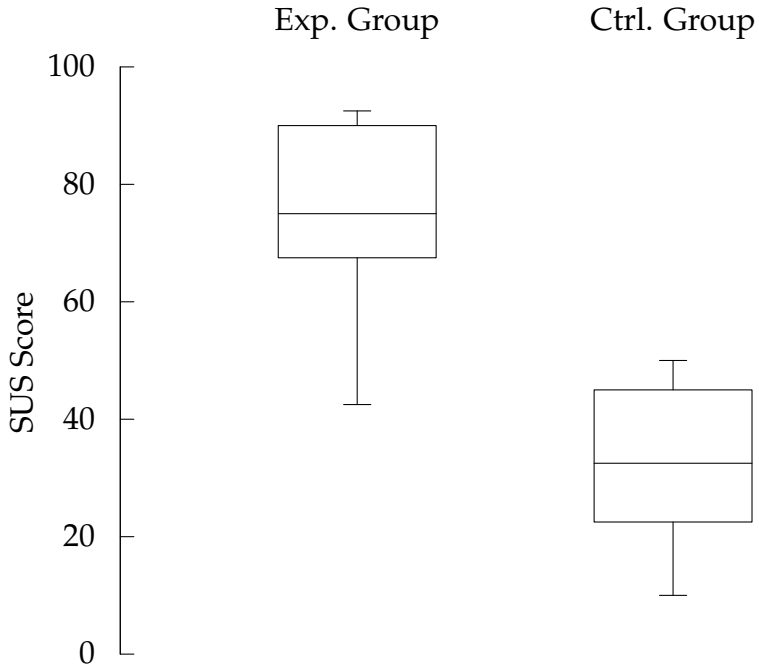


Figure 6.10: System Usability Score per Group

score obtained for HAWKSHAW lies clearly above the average. In fact, it converts to a percentile rank of 73%, which means that HAWKSHAW has a higher perceived usability than 73% of the other systems tested.

We also assessed the reliability of the SUS in our user study with Cronbach's Alpha—a measure widely used in social sciences to calculate the internal consistency of psychometric tests. High alpha values basically indicate that *"answers to a reliable survey will differ because respondents have different opinions, not because the survey is confusing or has multiple interpretations."* [Nor10]. In our case, we calculated a value of 0.937, which is commonly considered as an indicator for excellent internal consistency. This high alpha value is in line with those values that have been previously reported in the literature for the SUS [BKM08, LS09].

The MWU test rejects $H3_0$ at the 99% confidence level (p-value=2.95E-8). We therefore accept $H3$: the SUS scores are different for the experimental and control groups. As a consequence, we can now answer our last research question, RQ3: Yes, the perceived usability of HAWKSHAW is significantly higher than that of traditional tools.

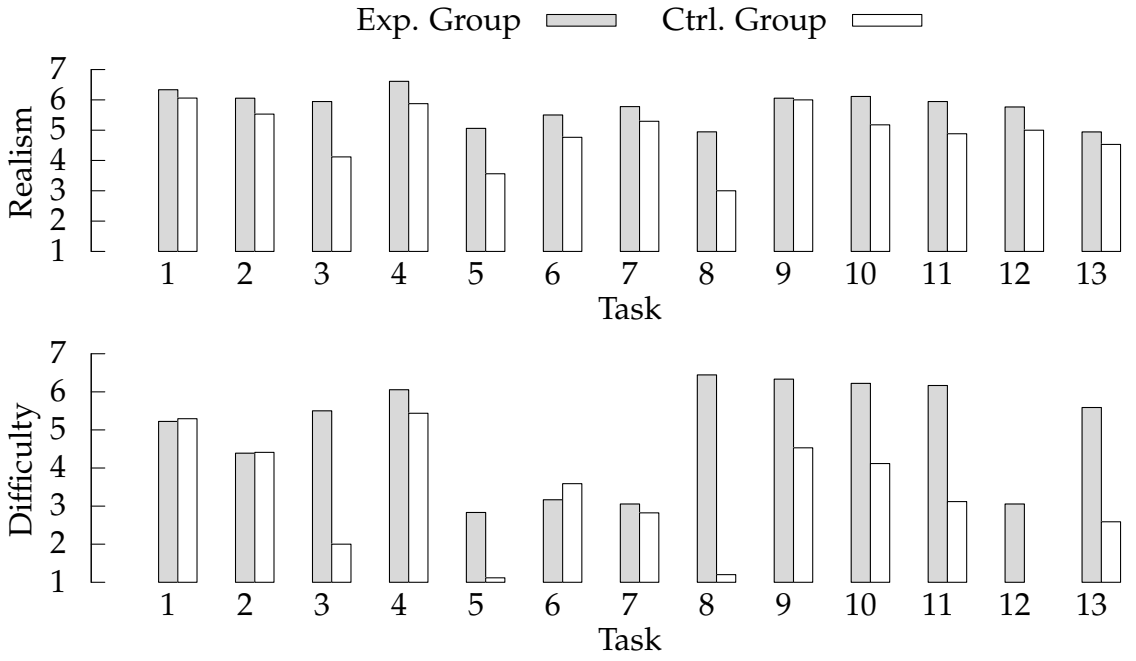


Figure 6.11: Realism (1 - very unrealistic, 7 - very realistic) and Difficulty (1 - very difficult, 7 - very easy) per Group and Task

6.4.9 Detailed Task Analysis and Interpretation

The overall results presented in Section 6.4.8 show a clear advantage of our approach over the baseline tools. To analyze where the advancements come from, we performed post-hoc tests for correctness and completion time for each task individually. Because of the post-study character of the SUS, we could not break down the overall system satisfaction into individual scores for each task. However, to still obtain individual results, we additionally surveyed the subjects after each task with the post-task SEQ questionnaire. Furthermore, we assessed the practical relevance of each task by asking the subjects whether they found the task realistic or not. An overview on the usability scores per task and group can be found in the lower chart of Figure 6.11, whereas the upper one shows how the two groups rated the practical relevance of each task.

Both the post-hoc tests for correctness and completion time as well as the tests to compare the perceived difficulty and realism levels were performed with the MWU test, unless stated otherwise. The visual analysis of the distribution of the data for the individual tasks suggested non-normality and therefore called for a

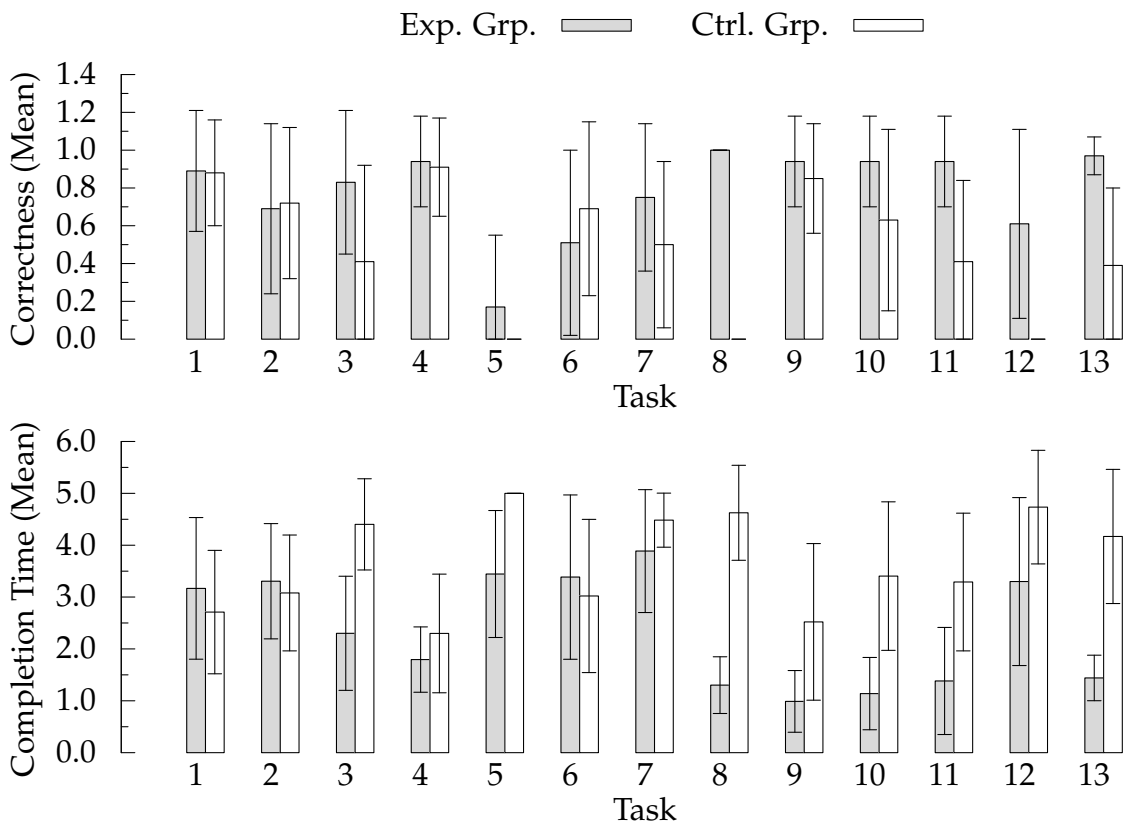


Figure 6.12: Mean Correctness and Completion Time per Group and Task. The Error Bars show the Standard Deviation.

non-parametric test. We applied the Bonferroni-Holm method for the post-hoc tests, which is a sequentially rejective version of the simple Bonferroni correction for multiple comparisons. It strongly controls the family-wise error rate at level alpha and *"ought to replace the classical Bonferroni test at all instants where the latter usually is applied."* [Hol79]. The global confidence level remained at 99%.

Table 6.4 lists the mean, mean difference, median, and std dev. values for the correctness and completion time in seconds per group and task. The differences that are statistically highly significant according to the MWU test are marked with two stars (**). Figure 6.12 depicts the results; it shows the mean correctness and completion time per group and task, as well as the corresponding std dev. values. What follows is a brief summary of the statistics obtained for each task, as well as our interpretation of these results.

		Correctness (Points)				*Completion Time (Min:Sec)*			
ID	Group[a]	Mean	Mean$_\triangle$[b]	Median	StdDev	Mean	Mean$_\triangle$	Median	StdDev
T1	exp.	0.89	+0.01	1.00	0.32	03:10	+00:27	03:08	01:22
	ctrl.	0.88		1.00	0.28	02:43		02:45	01:11
T2	exp.	0.69	-0.03	1.00	0.45	03:18	+00:14	03:04	01:07
	ctrl.	0.72		1.00	0.40	03:05		03:07	01:07
T3	exp.	0.83	+0.42	1.00	0.38	02:18	-02:06**	01:56	01:06
	ctrl.	0.41		0.00	0.51	04:24		05:00	00:53
T4	exp.	0.94	+0.01	1.00	0.24	01:48	-00:30	01:46	00:38
	ctrl.	0.91		1.00	0.26	02:18		02:10	01:09
T5	exp.	0.17	+0.17	0.00	0.38	03:27	-01:33**	03:01	01:13
	ctrl.	0.00		0.00	0.00	05:00		05:00	00:00
T6	exp.	0.51	-0.18	0.60	0.49	03:23	+00:22	03:25	01:35
	ctrl.	0.69		1.00	0.46	03.01		03:09	01:29
T7	exp.	0.75	+0.25	1.00	0.39	03:53	-00:36	04:13	01:11
	ctrl.	0.50		0.55	0.44	04:29		04:35	00:31
T8	exp.	1.00	+1.00**	1.00	0.00	01:18	-03:19**	01:09	00:33
	ctrl.	0.00		0.00	0.00	04:38		05:00	00:55
T9	exp.	0.94	+0.09	1.00	0.24	00:59	-01:32**	00:46	00:36
	ctrl.	0.85		1.00	0.29	02:31		02:18	01:31
T10	exp.	0.94	+0.31	1.00	0.24	01:08	-02:16**	00:59	00:42
	ctrl.	0.63		1.00	0.48	03:24		02:45	01:26
T11	exp.	0.94	+0.53**	1.00	0.24	01:23	-01:54**	00:57	01:02
	ctrl.	0.41		0.33	0.43	03:24		03:10	01:20
T12	exp.	0.61	+0.61**	1.00	0.50	03:18	-01:26**	03:17	01:37
	ctrl.	0.00		0.00	0.00	04:44		05:00	01:06
T13	exp.	0.97	+0.58**	1.00	0.10	01:26	-02:44**	01:23	00:26
	ctrl.	0.39		0.43	0.41	04:10		04.48	01:18

** *high significance (global $\alpha = 0.01$, Bonferroni-Holm method applied)*
[a] *experimental group = exp., control group = ctrl.*
[b] $Mean_\triangle = Mean_{exp} - Mean_{ctrl}$

Table 6.4: Individual Results per Task

Task 1: Learning about class hierarchies

Results. The difference between the experimental and control groups is statistically insignificant in both correctness and completion time. The subjects of both groups agreed that the the first task was easy to solve with the available tools and that the task was realistic—in both cases with insignificant differences in their ratings.

Interpretation. Displaying and navigating a class hierarchy is well-supported in Eclipse through its *Type Hierarchy View*. The usage of the view was explicitly explained to the control group in their tutorial. In light of this, it is remarkable how well HAWKSHAW competed with Eclipse, especially since most subjects were

already familiar with Eclipse, whereas none of them had used our tool so far. In addition, we intentionally stated the task description in a way that forced the subjects to reformulate the given sentence for HAWKSHAW. The goal was to test whether they could make use of the query composition guidance provided by our approach. In particular, the noun *"subclass(es)"* had been omitted[6] from our natural-language annotations and, instead, the question had to be entered as *"What classes extend...?"* or *"What classes inherit from...?"*. Unsurprisingly, a common complaint by the subjects of the experimental group therefore was that our approach did not directly allow for questions, such as *"What are the subclasses of...?"*. Despite this twist, 16 out of 18 subjects of the experimental group succeeded in quickly submitting a correct query. For unknown reasons, the other two persons wrote down only the (correct) package names while unfortunately omitting the class names. In such cases of doubt, we decided against HAWKSHAW—and in favor of the baseline tools—and counted the answers as incorrect.

Task 2: Finding methods that operate on an instance of a given class

Results. We could not observe any notable difference between the two groups, neither in terms of correctness and completion time, nor concerning their difficulty ratings. On average, the control group found it less plausible that they would have to solve such a task in practice. However, the difference to the experimental group is statistically insignificant.

Interpretation. Finding methods that use a given type as a parameter can be achieved in Eclipse in two ways: through the *References*-feature in the context-menu, or by using the *Java Search* dialog. Both features were illustrated with screenshots in the control group's tutorial. Nonetheless, the simplicity of our query interface was able to compensate for the head start of the baseline tools. Particularly for this task, we received ambiguous qualitative feedback on one feature of HAWKSHAW: our tool by design does not allow for fully qualified type or method names in questions. Instead, users have to reference Java entities simply by their identifier. While some subjects told us that they appreciated this feature because it resembles closely how they reference types and methods when they

[6] Meanwhile we added the concepts *Superclass* and *Subclass* to SEON. For any triple *"a extends b"*, a reasoner will now automatically classify *a* as *Subclass* and *b* as *Superclass*.

talk to colleagues, others were confused that they did not find the full qualifiers among the proposals provided during query composition. The resolution was straight-forward: we now support both short identifiers and fully qualified names.

Task 3: Finding the common callers of two methods

Results. The mean correctness achieved by the experimental group is roughly twice as high as that of the control group (0.83 vs 0.41). However, when performing the MWU test in conjunction with the Bonferroni-Holm correction, the null-hypothesis is retained with a p-value of 0.032 at a corrected α-level of 0.001. Hence, the difference is deemed insignificant. We therefore binned the answers into fully correct (correctness=1.0) and incorrect ones (correctness < 1.0) to be able to perform Pearson's Chi^2 test on the null-hypothesis that the correctness is independent of the subject's group. With a p-value of 0.001, the null-hypothesis is rejected at a confidence level of 99%. For completion time, the results are more conclusive: the experimental group was able to solve Task 3 significantly faster than the control group (MWU test, corrected α=0.001, p-value $<$ 9.69E-6). These results translate to those on realism and difficulty: the experimental group found the task very realistic and very easy to solve. The control group, in contrast, rated the degree of realism on average with about four (most subjects were undecided), whereas they perceived the level of difficulty as very high.

Interpretation. Finding code that invokes a given method is widely recognized in the literature, *e.g.*, by Sillito *et al.* in [SMV08], as a common task among developers and corresponding search features are incorporated in almost any modern IDE. However, we increased the difficulty level of the task by adding a conjunction: we asked for the common callers of two methods, instead of simply the callers of a single method. Hence, we tested the control group's ability to compose the information fragments of two Java searches, while the experimental group could simply chain the parts in one question. The difference in completion time and perceived difficulty is nonetheless surprising because, of the two methods given in the task, the second one had only two callers in the whole system—applying the right search strategy, the correct answer could be obtained in matter of seconds. However, we observed that multiple subjects of the control group only searched for the references of the first method and then browsed the code of the 15 results for invocations of the second one. This poor search strategy lead to the higher

completion times observed for the control group. The potential difference in correctness could be a result of failure to check all the callers of the first method within the allotted time. However, given the inconclusive results of the statistical tests, we refrain from drawing a final conclusion on the correctness in Task 3.

Although the composition of information fragments is an important field of research in software engineering [FM10], our task is artificially more challenging. It is however notable that we explicitly faced a similar question during a maintenance task, when we were checking for violations of a locking-related idiom to resolve a concurrency issue in HAWKSHAW.

Task 4: Learning about code ownership

Results. Both groups performed equally well with respect to correctness and completion time. They attributed the task a very high degree of realism and a very low level of difficulty.

Interpretation. The *History View* of Eclipse provided the control group with a quick and easy to read, table-based view on the change log of individual files. Also the experimental group faced no difficulties in reformulating the task description into a query understandable by HAWKSHAW. Notable was that two subjects mentioned that they looked at the proposals presented by our approach and were unsure about the difference between the verbs *"commit"*, *"modify"*, and *"change"* (*e.g.*, *"What developers change...?"*). We explained to them after the study that the three verbs are treated as synonyms—they are compiled into exactly the same SPARQL query—and that the purpose of the synonyms is to provide multiple ways of formulating a query, to achieve a similar level of freedom as with full natural language input. Their response was that they would have appreciated a short demo of HAWKSHAW. Nonetheless, both of the subjects performed reasonably well despite their limited knowledge of the details of our approach.

Task 5: Finding exceptional entities in terms of changes

Results. Both groups performed equally poor in solving Task 5. None of the control group and only three out of 18 subjects of the experimental group obtained the correct solution. The small difference between the groups lead the MWU test to retain the null-hypothesis concerning correctness. For completion time, the test

yielded a significant difference in favor of the experimental group. However, this was simply a consequence of the experimental group's premature assumption that they had provided the right answers. The difficulty of the task was perceived high by the experimental group and very high by the control group. The first group assessed the degree of realism as high, the second one as rather low. Both differences in ratings were statistically significant.

Interpretation. Succeeding with only the baseline tools required some familiarity with SVN's numbering scheme for revisions: with the Eclipse Subversive plug-in, the version numbers are displayed next to each directory or file name. Searching first for the directory with the highest number and then checking its files' version numbers would have quickly yielded the correct answer. To obtain the correct answer with HAWKSHAW, one had to enter, *e.g.*, the question *"What file has the most versions?"* Interestingly, nine out of 18 subjects of the experimental group gave the right answer to the wrong question, *i.e.*, they wrote down the file that was changed by the most developers instead. The reason for this misunderstanding was that they relied too much on the guidance provided by our approach. When they wanted to enter *"What file changed...?"*, HAWKSHAW instead proposed *"What file is changed by..."*. The only way to complete the latter sentence was by choosing *"...the most developers?"*. Instead of questioning the result and then reformulating their information need, they accepted the non-applicable answer and proceeded to the next task. We attribute this shortfall to the study setting and claim that, in a real setting, a developer would not be satisfied with the answer and further strive for an appropriate answer.

Task 6: Learning about the changes of other team members

Results. HAWKSHAW performed as well as the baseline tools in Task 6, with no significant differences according to the results of the MWU test. The mean correctness of the control group was observed slightly higher than that of the experimental group. We therefore, again, binned the answers into correct and incorrect ones, and ran the Chi^2 test on the data. The test this time clearly retained the null-hypothesis at a confidence level of 99% with a p-value of 0.229. The difficulty was perceived rather high in both groups, whereas the realism was rated rather high, with an insignificantly lower rating among the participants of the control group.

Interpretation. We were surprised to see that the control group could not outperform the experimental group significantly in this task. Our expectations were grounded on the development history of Apache Ivy and the concrete developer we named in the task description. When one opened the Eclipse *History View* on the project, the author of the five most recent commits (which were therefore displayed on top) was the one we were looking for and the corresponding files could be found by clicking through the first few table items. In contrast to the baseline tools, solving this question with HAWKSHAW was particularly difficult because the subjects first had to obtain all the files changed by the particular developer. They then had to sort the files by their modification dates in the result view. Multiple participants therefore later said that they did not understand how to sort the results properly and would have wished for a thorough example in the tutorial related to this functionality.

Task 7: Becoming familiar with a team member's work

Results. No statistical differences in correctness and completion time between both groups could be observed for this task. Most subjects found the task realistic but rather difficult to solve.

Interpretation. Similar to the last task, this task was concerned with team awareness—but this time from an issue-perspective. Mapping the task description to a search strategy asked for some creativity because there is no direct way to obtain the feature request implemented by a given developer. However, finding any closed feature requests with the appropriate assignee provides a good approximation, which can then be verified against the change log. The control group had to use a tool external to the IDE to solve the task: the Web front-end of JIRA. The simple search of JIRA provides an input mask to query explicitly for feature requests with a given assignee and resolution, rendering the task straight-forward. The experimental group had to reformulate the task to a query containing an adjective, *i.e.*, *"What **closed** feature requests are assigned to...?"*. We did not mention the possibility of using adjectives in the HAWKSHAW tutorial, yet most subjects of the experimental group were able to obtain the correct answer.

Task 8: Finding issue reports relevant to other team members

Results. None of the subjects of the control group were able to solve the task correctly, whereas those of the experimental group succeeded without exception in under two minutes. As a consequence, the control group found the task very difficult and rather unrealistic—in contrast to the experimental group, which rated the same task significantly different: very easy and rather realistic.

Interpretation. The task was neither solvable with the JIRA Web front-end, nor from within the IDE. However, the control group could have run a straightforward Google query to obtain the correct answer within seconds. It seems that they did not think of this possibility and tried instead to submit queries with the advanced search of JIRA until the time for the task ran off. The HAWKSHAW group, on the other hand, did not report any problems.

The ratings on the relevance of the tasks show some disagreement. It has to be mentioned however, that the task was derived from the literature [FM10] and that there exist JIRA plug-ins, as well as a feature request for JIRA itself with more than 50 votes to support this particular information need. We therefore consider the task relevant, although the control group disagreed with us.

Task 9: Finding issues affected by a particular blocker

Results. While correctness was not an issue for either group, the subjects with HAWKSHAW were significantly faster than those using the JIRA Web front-end. Both groups found the task very realistic, but there was a statistically significant difference in the perceived difficulty.

Interpretation. The control group was about three times slower than the experimental group. The difference is worth emphasizing because the Web front-end of JIRA contains a clearly visible section *Issue Links* near the top of the overview page of each issue. There, the issues blocked by the current one (including the word *"blocked"*) are listed. On the other hand, the excellent performance of the experimental group shows how well the subjects adapted to our quasi-natural language interface after only a few tasks.

Task 10: Finding the Java classes affected by a particular issue

Results. For the tenth task, we again observed a significant difference in completion time between the two groups. The difference in correctness was insignificant for the MWU test. The Chi^2 test retained the null-hypothesis that the number of fully correct answers depends on the group at a confidence level of 99% but rejected it at 95% with a p-value of 0.012. The difference observed for the ratings concerning realism was insignificant with MWU at a confidence level of 99%. However, it is significant at 95% (p-value=0.027). For the difficulty ratings, we observed a significant difference, even without adjusting the α-level.

Interpretation. JIRA integrates with SVN to show the files that were committed to resolve an issue. This functionality was illustrated in the tutorial given to the control group. The file diffs could then be viewed online by the subjects to verify whether the changes really applied to the top-level class declared within that file. Despite this well-integrated approach, HAWKSHAW outperformed the baseline tools by far with regard to completion time. The statistical results on correctness are inconclusive, but slightly in favor of HAWKSHAW when we incorporate Chi^2 with a confidence level of 95%.

Task 11: Finding all the issues affecting a particular Java class

Results. The task was solved faster and with a higher correctness by the experimental group than by the control group. This advantage is statistically significant. The control group found the task rather realistic on average, but rather difficult to solve. The experimental group was convinced that the task is very realistic and that it is very easy to solve with our approach. The difficulty ratings are significantly different. For realism, the differences are significant at a confidence level of 95%, but not at a level of 99% (p-value=0.03).

Interpretation. The subjects of the control group had to manually search through the SVN log of the class for issue numbers in the commit messages. Their difficulty rating indicates that this is a tedious task—and also error-prone, as it is easy to miss a reference to an issue. We generally noticed that the subjects often provided incomplete solutions with the baseline tools, whereas those in the experimental group in most cases provided either no solution or a fully correct one. This is reflected in the many high median correctness values of 1.0 listed in Table 6.4. The effect is also apparent in Figures 6.13 and 6.14, where the stacked bar charts

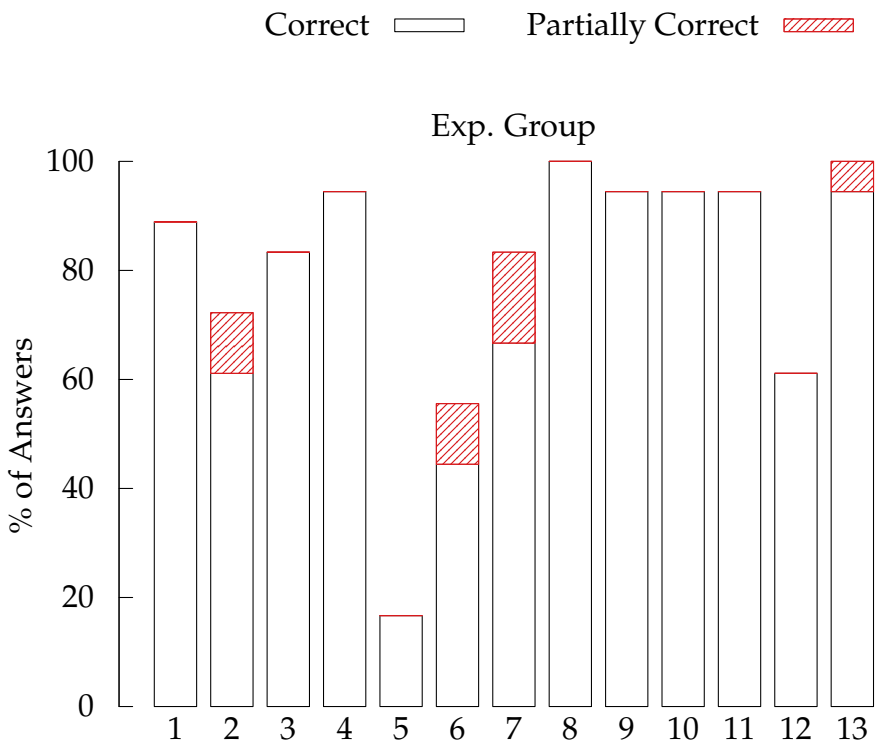


Figure 6.13: Percentage of correct and partially correct Answers given by the Experimental Group for each Task

show for both groups and for each task the correct and partially correct answers in percent of the total number of answers we received. The left chart shows the results for the experimental group; only in four out of 13 tasks, partially correct answers were given, as opposed to the control group, shown in the right chart, where partially correct answers were given in nine out of 13 tasks. The observed phenomenon is grounded in the nature of our approach: if users are able to correctly formulate their question, then recall will most likely be at 100%.

Task 12: Finding exceptional entities in terms of error-proneness

Results. No participant in the control group was able to solve Task 12 correctly in time, which corresponds to the difficulty ratings. The experimental group performed reasonably well in terms of correctness and completion time, but still perceived the task as difficult. The groups, however, both agreed that the task was

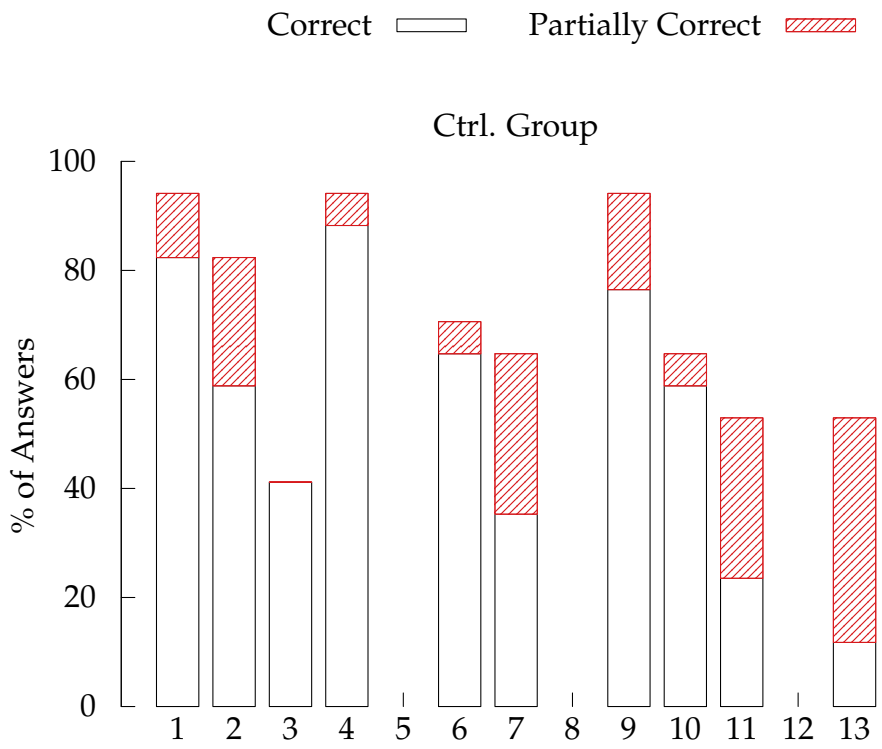


Figure 6.14: Percentage of correct and partially correct Answers given by the Control Group for each Task

realistic: the experimental group found it very realistic, the control group rather realistic. The difference between their difficulty ratings is significant; the control group found the task more difficult to solve than the experimental group.

Interpretation. One difficulty that participants from both groups reported was that they did not understand the term *"error-prone"* as non-native English speakers. A few users of HAWKSHAW were thus not able to translate the task description into a meaningful question with a subject-predicate-object structure. The baseline tools did not provide any means to quickly solve the task, but rather it was necessary to skim the SVN log of the whole project for issue numbers and associated files. This was rather unrealistic in the short amount of time given to the subjects. Despite the unfavorable situation for the baseline tools, we can still conclude that the users of HAWKSHAW were able to solve a task reasonably well that was deemed important by the majority of the participants of our study, as well as by other researchers [KZWJZ07].

Task 13: Finding issues that affect multiple Java classes

Results. Except for one outlier, the maximum correctness was achieved by all subjects of the experimental group. Nine out of 17 subjects of the control group either provided a correct answer or at least a partially correct one. On average, the control group nearly maxed out the available time, whereas most members of the experimental group were able to solve the task in less than one and a half minutes. Statistically significant differences between the groups were also observed for the difficulty assessment: the experimental group found the task easy to solve, the control group rather difficult. Both groups agreed that the task was rather realistic. Because Task 13 was very similar to Task 11, we tested whether there was a difference in the performance per group between the tasks. For the experimental group we determined with the Related-Samples Wilcoxon Signed-Rank test that there was no significant increase in completion time (p-value=0.396, α=0.01) between Task 11 and 13. For the control group, the increase was significant—at least at the confidence level of 95%, with a p-value of 0.038. We observed no significant difference in correctness for either group.

Interpretation. To conclude our study, we re-iterated on a previous task, Task 11, but asked for the intersection among the answers to two distinct questions this time. The SVN logs that the subjects of the control group had to go through were comparable in size for both tasks, but for Task 13, the number of relevant commit messages doubled because two, instead of one, Java classes were involved. As we can see from the results, the influence of the additional twist on the correctness was negligible for both groups but there was a negative impact on the completion time for the control group only. Because the composition of different information fragments happens implicitly and therefore automatically in HAWKSHAW, users with HAWKSHAW were much faster than those with the baseline tools.

6.4.10 Summary of Results

The overall results presented in Section 6.4.8 showed that with HAWKSHAW, users were able to solve the body of tasks in our study with a significantly higher correctness in a significantly shorter amount of time. In Section 6.4.9 we have analyzed which tasks contributed the most to the superiority of our approach.

The analysis of the individual tasks showed that, for the Tasks 5, 8, and 12, none

of the subjects were able to provide a fully correct solution with the baseline tools. As a consequence, they used up the total amount of time allotted to that task. To test whether only these three tasks were responsible for the significant difference in correctness and completion time, we excluded them from our data and re-ran the MWU test for the adjusted overall results at a confidence level of 99% with an α of 0.01. The null-hypothesis concerning the correctness was nonetheless rejected with a p-value of 0.0048; similarly the null-hypothesis for completion time, but with a p-value of 1.47E-5. In Tasks 1 to 7, both groups achieved a similar correctness. It was possible for the control group to obtain good scores in terms of correctness in the individual tasks, which suggests that our selection of tasks in general did not penalize the baseline tools unduly.

For five tasks (Tasks 3, 9, 10, 11, and 13) we could observe a significant reduction of completion time when HAWKSHAW was used, rather than the baseline tools. In some cases, the average time savings were as high as 300%. Furthermore, the experimental group was able to solve Task 8 and 12 within the time limit of five minutes, whereas the control group was not. Unsurprisingly, the largest time savings comes from the cross-domain tasks, but HAWKSHAW also yields a significant benefit where the composition of data from different tools or multiple queries is crucial for solving the task at hand quickly. We emphasize that our approach performed as good as the mature code search features of Eclipse.

Besides these gains in efficiency, another main advantage of our approach lies in the significant usability improvements reported by the subjects. The individual responses to the post-task usability questionnaire SEQ support the excellent SUS score achieved by HAWKSHAW: The subjects of the experimental group perceived eight tasks significantly easier than the members of the control group (*i.e.*, Tasks 3, 5, 8, 9, 10, 11, 12, 13). The remaining five tasks were rated similarly by both groups—including the ones related to source code search. That our approach stands up against Eclipse in the source-code-centric tasks not only in terms of completion time but also in terms of usability is especially remarkable because it were possible to solve the tasks in Eclipse with only two mouse-clicks or by pressing a single shortcut.

We are convinced that we have not yet fully exploited the potential of the HAWKSHAW approach: with more sophisticated grammar rules and additional synonyms encoded into our ontologies, there is even more room for improvement.

6.4.11 Threats to Validity

Internal Validity is concerned with uncontrolled factors that may have biased our results in favor of either the experimental or the control group. One concern is that the subject's experience level might have been distributed unevenly over the two study groups. To mitigate this threat, we have randomly assigned the participants to the experimental and control group. Figures 6.6 and Figure 6.7 illustrate further that this concern is unjustified, *i.e.*, that the distribution of experience and skills was fair.

For measuring completion time, we provided the participants with an experiment timer application as part of the Eclipse IDE. Since each participant himself was responsible for reporting time correctly (*i.e.*, starting and stopping the timer, as well as writing down the number of elapsed minutes and seconds after each task), it is possible that incorrect or even completely fictitious task times have been reported by some of the subjects. However, we carried out inspections at random and, in principle, the design of our study did not motivate the participants to game the system.

Lott and Rombach mention that having a subject perform several related tasks within a short time interval may cause non-negligible learning effects [LR96]. Learning effects may have affected the results on correctness and completion time because the experimental group could rely solely on HAWKSHAW for each of the task, whereas he subjects of the control group had multiple tools at their disposal. Depending on the task, they had to use of a different tool to obtain the correct answers and could therefore benefit less from potential learning effects. While the exact strength of the potential learning effect remains unknown, we argue that the related threat to validity is relatively low for Tasks 1 to 6 and 10 to 13. For the first six tasks, the participants of the control group benefited from several months or even years of experience in using the Eclipse IDE, including the Java search features and the history view. The users of HAWKSHAW, however, had never used the quasi-natural language approach before. In Tasks 10 to 13, the subjects of the control group could reuse the same tools again and in a similar fashion as in the previous tasks. The cross-domain nature of these tasks simply required an additional step to compose the partial results obtained from each tool. Learning effects might have had a stronger impact on the results for Tasks 7 to 9, which were related to bug and issue management. The participants knew the concept of

issue trackers in principle, but barely any of them had worked with the JIRA issue tracker before, and the tutorial we provided them could only account for this lack of experience to a certain extent.

Regarding *construct validity*, it is possible that the questionnaires used to measure usability are not an adequate means for assessing usability of the baseline tools or our approach. Given that the SUS and SEQ were used successfully in thousands of different usability assessments in various application domains and that numerous researchers attribute an excellent reliability to it, we believe that this threat is relatively low. Furthermore, we calculated a very high Cronbach alpha value for the SUS scores observed in our study, indicating that the SUS measured consistently. It has also been shown that the SUS does not provide a strong correlation to task-level metrics. Task completion rates, for example, explain only around 5% to 6% of the changes in SUS scores [SL09]. This leaves us confident that we were able to measure overall system satisfaction mostly independent from the choice of our tasks.

Threats to *conclusion validity* may arise from too few participants in our study. To alleviate this threat—and since we could not safely assume a normal distribution for our data—we selected the non-parametric MWU test which also works on non-normally distributed data and is robust even against small sample sizes. We complemented the MWU test with a thorough visual analysis in box plots, histograms, Q-Q plots, and where appropriate, with other statistical test procedures.

External Validity is the degree to which the results can be generalized and transferred to other situations. In that respect, our subjects may not have been a representative sample for junior developers employed in industry. However, we only invited students that were studying in the fourth semester of the Bachelor's level or higher. Thus they had successfully passed multiple courses on programming, algorithms and data structures, as well as on software engineering methodologies. Many of them had already completed an internship in a software development company and only a few more courses plus their Bachelor's thesis were left before they would begin their careers in industry.

None of the participants were familiar with Apache Ivy—the software system we used in our study. Developers with more knowledge of the system might work differently or be able to apply more effective strategies to use their existing tools. The situation that participants faced in our study, however, is comparable to that of developers newly joining a software project.

We only performed our study with one object-oriented system, which may not be representative for the family of object-oriented systems in industry. Especially for the cross-domain tasks, we strongly relied on the rigid change process of the Apache Foundation, in order to be able to reliably link issues to changes. In projects for which no such software evolution data is available, the completeness of the results returned by HAWKSHAW will suffer and consequently, its users will perform less effectively. However, in such a scenario, also the users of the baseline tools will be challenged. Another selection of baseline tools may provide more competitive features and a higher usability than the one we have chosen.

The tasks chosen for our study might not be realistic or they might favor HAWKSHAW and therefore the results might not generalize to others than the tools used in the study. We re-used existing catalogues of common developer questions that were compiled by other researchers through surveys and interviews of practitioners, but we had to adapt most questions to make them more concrete. We also extended some of the original questions to assess certain facets of our approach, and thus deviated further from the original questions. To control the effects of our modifications, we asked the subjects to assess the practical relevance of our tasks. Most tasks were clearly considered by both groups as being realistic. For the few instances where the subjects were at odds with each other, we provided a thorough discussion on why we still are convinced of the relevance of our selection. However, the subjects' ratings concerning practical relevance still need to be treated with caution: we detected a modest correlation ($r = 0.4$) between difficulty and practical relevance, *i.e.*, those subjects that rated a task more difficult also tended to attribute a lower practical relevance to it. The exact causal chain remains unknown and further investigations are necessary, but it is possible that the practical relevance ratings were confounded by difficulty and therefore cannot support the generalizability of our approach.

The baseline we compared HAWKSHAW against was comprised of a common IDE with SVN support, the Web front-end of a widely used issue tracking tool, as well an internet browser to access the issue tracker and to perform Web searches. Our selection of tools might not have taken into account that development teams in industry might use even more powerful programs.

6.4.12 Discussion and Synthesis

The user study focussed on less experienced subjects since we expected our approach to be especially apt for novice developers, not yet deeply familiar with the tools available. The reasoning behind this was that novice developers will most likely not know about most of the advanced features of an IDE. A single point of access for their information needs would relieve them from browsing all menus and dialogs to find what they are looking for, from reading help pages, and so on. The positive feedback received from some of the more seasoned participants provides anecdotal evidence that the results of the study can be generalized to a broader target group of users. Further investigations are needed to support this claim with scientific evidence.

The HAWKSHAW framework revolves around a knowledge base built with Semantic Web technology and a quasi-natural language interface. The Semantic Web yet struggles to find a wide adoption in the field of software evolution research, whereas, for example in life sciences, many applications have demonstrated the value of the Semantic Web for processing and sharing large corpora of information (*e.g.*, in [KSG+10]). The same accounts for natural language interfaces, which have been mostly neglected in software evolution research so far, but recently gained momentum in other domains. Popular examples are Apple's Speech Interpretation and Recognition Interface (Siri),[7] the Wolfram Alpha answer engine developed by Wolfram Research,[8] and IBM's Watson computer system for answering natural language questions [Fer11].

The overall conclusions we can draw from the strong empirical results found in our user study is that both the Semantic Web and natural-language interfaces exhibit significant potential for building the next generation of software engineering support tools. Such technology should therefore be at least considered whenever researchers in the field of software engineering devise approaches that involve knowledge representation and developer-computer interfaces. A quasi-natural language interface such as the one incorporated in the HAWKSHAW framework requires basically no learning effort from its users and therefore can accelerate the adoption of novel research tools in practice.

The design of our study and particularly the task selection take into account the

[7] http://www.apple.com/iphone/features/siri.html
[8] http://www.wolframalpha.com/

two aspects mentioned above: we evaluated the quasi-natural language interface of HAWKSHAW against different classical user interfaces, but we also also looked at the advantage of the integrated view of our knowledge base over an heterogenous tool landscape where users are exposed to multiple sources of information.

The first three tasks of our study were selected to compare HAWKSHAW with the Java search of Eclipse, which is based on a classical, context-menu-driven user interface. Remarkably, users were as effective with the natural language interface as the with the code search, although the latter is much more specific and therefore highly optimized to these kind of tasks. While there is no indication that in replacing the current code search with HAWKSHAW we would gain benefits in terms of effectiveness for simple code search tasks, HAWKSHAW can still complement the existing features by providing an entry point to developers not yet familiar with the IDE. Furthermore, a query-language-based approach such as ours provides additional expressiveness when it comes to solving more complex tasks. Thus it can relieve users from manual composition of the results obtained from multiple invocations of a menu-driven search. This claim is supported by the results for Task 3 on time efficiency.

A similar observation can be made for the tasks related to the development history where users of the baseline tools had to browse a table-based view. Such views sometimes implement simple keyword-based search and basic filtering, but these search widgets usually lack of means to express relationships between data. In our study, users of the baseline tools consequently spent much more time for going through each of the entries of the table view than the users of HAWKSHAW, who generally succeeded in clearly specifying their information needs with a concise query.

Subjects of the control group had to use the Web-front-end of the issue tracker for solving the Tasks 7 to 9. The front-end provided Web forms for querying and displayed the results as hypertext documents. Many subjects of the control group overlooked very prominent information, such as the blockers of certain issues, which suggests that they were overwhelmed by the sheer amount of information displayed. This may be attributed to their unfamiliarity with JIRA and users are likely to overcome these issues after a training phase, but yet the advantage of HAWKSHAW lies in its high learnability—users do not need to familiarize themselves with another user interface paradigm when working with bug reports, other than the one they already use for searching in code, querying the development

history, and so on. Notable are also the problems that subjects faced when answering Task 8 where they had to find the issues on which two particular developers had commented in the past. JIRA did not explicitly support searching for issues a person has commented on, so the participants had to fall back to a regular Web search for obtaining the correct answer. With the HAWKSHAW framework, there is no need to implement such specific searches. Instead, data is simply described through an OWL ontology, along with the general grammatical structure of the queries, and consequently even queries will work that were not explicitly foreseen. Since describing data with OWL is not much different from defining a relational database schema, the overhead of doing so is negligible compared to classical approaches.

Whereas the previous tasks focussed on comparing different user interface paradigms against HAWKSHAW, the last four tasks put emphasis on the advantage that HAWKSHAW's integrated knowledge base yields over multiple, poorly integrated sources of information. The clear results for these tasks indicate that the manual composition of different information fragments is laborious and error-prone, and that our approach can provide significant relief in this regard. In synergy with the quasi-natural language interface, the knowledge base becomes a powerful tool even for novice users. That means they get a query language whose expressivity is comparable to formal query languages but overcomes the initial hurdle of learning a specific syntax and vocabulary.

The HAWKSHAW approach is not without its limitations. We demonstrated through our study that a surprisingly small set of static grammar rules and synonyms allows for a variety of different queries. However, additional investigations are needed to identify variations in the exact phrasing of conceptual queries that might occur when software engineers formulate their information needs in practice. The findings then need to be encoded in terms of static grammar rules. Furthermore, the natural language annotations (synonyms) of SEON are based on our personal vocabulary. This vocabulary might be biased towards the programming languages and tools we regularly use and therefore fail to adequately describe the concepts which developers with a different background are familiar with. In this context, we evaluated the use of general-purpose lexical databases of English, in particular the WordNet database [Mil95], to increase the vocabulary of HAWKSHAW with additional synonyms. However, for our approach, such databases have proven themselves unsuitable. The problem we encountered was

that many technical terms also have non-technical meanings in daily life. For example, the term "Method" from in object-orientated programming has synonyms such as "adjustment," "approach," "fashion," etc. If we automatically add those to the list of proposals presented by our query interface, then the developers are no longer restricted to reasonable questions, *i.e.*, they can then enter completely meaningless ones such as *"What fashion invokes the approach foo()?"* Sridhara *et al.* have reported similar issues when they applied linguistic tools to source code and other software artifacts [SHPVS08]. The authors propose to augment WordNet with relations specific to software, which could also be valuable for improving the HAWKSHAW approach further.

Currently, HAWKSHAW is not particularly well-suited for answering questions related to time intervals or specific points in time. For example, questions such as *"What classes were changed yesterday?"* or *"What bugs were fixed between May and August?"* cannot be entered directly. However, it is possible to query, *e.g.*, for all bug fixes and then sort the results by their fix date. This puts HAWKSHAW on a par with many tools used in practice, but formal query languages would have an edge over our approach in that respect (at the cost of additional learning effort).

While it is notable that existing catalogues of common developer questions rarely contain examples such as the ones mentioned above, we still know from our own experience that they occur frequently in practice, so that an adequate support is desirable. While the static grammar of HAWKSHAW can be extended to incorporate corresponding natural language rules, more research is needed to come up with an appropriate translation into SPARQL. Approaches such as Temporal Reasoning could provide a solution to this issue [Tap11].

Currently, textual or keyword-based search is unsupported because of the guided nature of the query composition approach. In consequence, users cannot search, for example, for all the files that contain the word "database." This is sufficiently well supported by existing tools so that we, in principle, see no immediate need for action. However, to preserve the idea of a single point of access for common information needs, it would still make sense to add special non-terminal symbols to the static grammar rules that would temporarily switch the query interface from a guided mode into one that allows for entering free-form text (*e.g.*, when entering opening quotation marks until closing ones are typed). The translation into SPARQL is then straight-forward thanks to built-in regular expression support of the language.

6.5 Related Work

LaSSIE was an early attempt to integrate multiple views on a software system in a knowledge base [DBS91]. It also provided semantic retrieval through a natural language interface. Frame systems, a conceptual predecessor to the ontologies of the Semantic Web, were used to encode the knowledge. The aim of LaSSIE was to preserve knowledge of the application domain for maintainers of the software system. HAWKSHAW does not yet incorporate any application-specific knowledge but focuses on answering common developer questions related to software evolution.

Hill *et al.* [HPVS09] presented an algorithm to extract noun, verb, and prepositional phrases from method and field signatures in source code to enable *contextual searching*. The queries they support are closer to keyword search on identifiers found in source code than to full natural language questions and they do not cover structural information, such as caller-callee or inheritance relationships among source code entities. In contrast to ours, the approach completely neglects history or bug and issue related queries.

Another promising approach for querying source code with natural language queries was introduced by Kimmig *et al.* [KMM11]. Their query interface uses part-of-speech tagging and stemming to enable free-form queries, while our approach guides developers during query composition and does not rely on any natural language processing. In consequence, it is possible to enter queries, which are not understood by the approach of Kimmig *et al.*—unlike HAWKSHAW, which prevents its users from composing unrecognizable queries. Furthermore, we support querying of multiple facets of software evolution, whereas Kimmig *et al.* did not report whether their approach can be generalized to domains other than that of static source code.

Many approaches have been proposed that use specific languages to query software artifacts. They are either based on standard database languages, such as SQL or Datalog (*e.g.*, CodeQuest [HVdM06]), customized Prolog implementations (*e.g.*, JQuery [JV03] or ASTLog [Cre97]), or a custom language (*e.g.*, SCA [PP96]). Their aim is to help developers in effectively exploring and better understanding code, uncovering information that would be impossible or extremely hard to find with standard tools. However, most of them require the user to master syntax and vocabulary of a specific query language. Our approach guides developers in

vocabulary, as well as in syntax, to construct well-formed and coherent questions about different aspects of a software system. To the best of our knowledge, most of these works are limited to source code queries and no other approach exists that supports multiple software evolution facets.

The *Sphere Model* by De Alwis and Murphy [dAM08] enables composition of different sources of information to implement conceptual queries. In our previous work, we have shown that HAWKSHAW is able to answer their queries that are related to static source code information and, in addition, our approach allows developers to formulate their questions with quasi-natural language. Thanks to the recent improvements of our approach, we can now also answer a broader range of evolution-related queries.

Fritz and Murphy [FM10] presented the *Information Fragment Model* that, similar to the previously mentioned *Sphere Model*, supports the composition of information from multiple sources, as well as the presentation of the composed information. However, the model does not support explicit querying, but rather allows for the ad-hoc combination of results from two queries obtained with other approaches, based on identifier or text matching.

The work by Kaufmann and Bernstein presented in [KB10] by is unrelated to the field of software engineering research. However, they presented a usability study of query interfaces with 48 users. The study incorporated geographical data encoded in an OWL knowledge base and four query interfaces featuring four different query approaches. The goal was to demonstrate the usefulness of natural language interfaces for casual end-users. One of the evaluated interfaces was Ginseng, which our approach was originally based on. The conclusion drawn from the experiment was that, with natural-language questions, *"users can communicate their information need in a familiar and natural way without having to think of appropriate keywords in order to find what they are looking for."* The authors also found empirical evidence that *"people can express more semantics when they use full sentences and not just keywords."* The results from the HAWKSHAW user study suggest that these insights are generalizable from geographical data and casual end-users to both software developers and the domain of software evolution and maintenance.

6.6 Conclusions

Nowadays, the sheer scale of many industrial software development projects demands a wide range of tools to support processes and enable collaboration. Mastering these tools to such an extent so that one can answer common information needs that arise during daily development tasks is challenging and puts a high cognitive load on developers.

In this paper, we have shown that quasi-natural language interfaces provide a valuable alternative to menu-driven search with modern IDEs and project tracking tools. We argued that our HAWKSHAW approach is helpful in solving various tasks related to software evolution and maintenance, and that it also scales to real industrial-size software systems. To support our claim, a user study with 35 subjects was conducted. In summary, the results of our study provide empirical evidence that:

- Overall, developers achieved a significantly higher correctness when solving common software evolution tasks with our quasi-natural language approach than with a baseline of more traditional tools, such as a common Java IDE and the Web front-end of a popular bug and issue tracker. Looking further at the results for each task individually, we observed that users of HAWKSHAW always achieved at least the same level of correctness as their counterparts with the baseline.

- Our approach leads to a significant improvement in time efficiency. Overall, we have seen time savings of 35.41% when compared to the baseline, with gains of up to 300% in some individual cases.

- The overall system satisfaction of users with HAWKSHAW is clearly better than that of the baseline's users. The subjects rated our approach on average with a total score of 74.31 on the System Usability Scale, whereas the baseline only achieved an average rating of 38.24. The high score of HAWKSHAW is directly related to its high usability and learnability.

Our approach serves as a single point of access to facts about source code and various other knowledge which is otherwise hardly integrated and locked away in project trackers or version control systems. Because it is based on quasi-natural language, getting familiar with the HAWKSHAW interface requires little

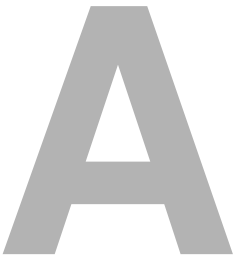
to no learning effort. It can be easily extended with additional grammar rules, synonyms then available to developers during query composition, and even with whole new software engineering domains—solely by using standardized means of knowledge engineering.

Future work will focus on the extension of HAWKSHAW's querying capabilities through the means described above, as well as on conducting a field study with professional developers in industry. From that, we hope to gain deeper insights on the potential and limitations of our query framework in practice.

Acknowledgements

The work presented in this chapter was supported by the Swiss National Science Foundation as part of the "Systems of Systems Analysis" (200020_132175) project. The authors are grateful to Serge Demeyer, Matthias Hert, and the anonymous reviewers of the ACM TOSEM Journal for the many thoughtful comments and the constructive feedback that greatly helped to improve the original paper.

Questionnaires



In the following, we list the two questionnaires including the instructions presented to the participants during the user study that we carried out to evaluate our approach. The first questionnaire was given to the experimental group, *i.e.*, those participants that evaluated the HAWKSHAW tool, whereas the second one was presented to the control group which could solely rely on the official Eclipse tools and a Web browser.

Evaluation of Hawkshaw

Experimental Group – Natural Language Interface

Department of Informatics, University of Zurich

27. April 2012

Dear Participant

The goal of this experiment is to evaluate HAWKSHAW, a query system for software evolution data, in comparison to more traditional software engineering tools. We appreciate your time very much—your support in this experiment is invaluable for the outcome of our research.

In the following, you will be asked to answer a set of questions about *Apache Ivy*, a popular open source Java tool for managing project dependencies. The questions will be about its source code and development & defect history, and each question has one or several answers. You belong to the experimental group and will answer the questions with our Hawkshaw natural language query interface.

You are asked:

- not to consult any other participants throughout the course of the experiment.
- to use only the tools and features listed in the beginning of each section.
- to solve all the tasks in their given order by filling in the requested information.
- to solve the tasks as fast as possible. For each task with time restriction, the available time is given in parentheses. Do not use more than the amount of time assigned to each task. In case that you cannot finish a task within the allotted time, you have to proceed to the next one.
- to write down how much time you have spent on each of the tasks (including the time needed for writing down the answer) in the text boxes printed at the end of the tasks (even if you could not complete a task). Use the *Experiment Timer* view to keep track of time. For the questions concerned with your personal details in the next section, there are no time restrictions—neither are there any for the debriefing section at the end of the questionnaire.

- to quickly asses the practical relevance (*i.e.*, how realistic is the task?) and difficulty of each task by checking the appropriate checkbox. By "realistic", we mean how likely you think that you would be interested to answer such a question in your daily work as a developer.

For any questions where full sentences are required from you, you can use either English or German in your answers—what ever you are more comfortable with.

Thank you for participating in our experiment,
Michael Würsch and Colleagues

☞ Please read now the Hawkshaw tutorial: `doc/hawkshaw-tutorial.pdf`

Personal Details

We kindly ask you to share below some personal information about you. It is used exclusively for statistical purposes and will not be traced back to specific individuals.

About yourself

1. How old are you? ..... years.

2. What is your gender?

   ○ Female

   ○ Male

Your experience

3. What is your current occupation?

   ○ Bachelor's student

   ○ Master's student

   ○ PhD student

   ○ Practitioner

   ○ Other: .....

4. How many years of development expertise do you have (any language)? ..... years.

5. How many years of Java development expertise with Eclipse do you have? ..... years.

Please tells us about your skills in the following areas:

		n/a	poor	average	good	excellent
6a.	My English skills are	○	○	○	○	○
6b.	My programming skills are:	○	○	○	○	○
6c.	My Java skills are:	○	○	○	○	○
6d.	My familiarity w/ *Jira* is:	○	○	○	○	○
6e.	My familiarity w/ *Subversion* is:	○	○	○	○	○

☞ Please start tracking time with the Experiment Timer when you begin with the next task

Questions concerning Program Code

> *Apache Ivy* is an object-oriented system implemented in Java. It is comprised of approx. 50'000 lines of code in 64 packages and 668 classes. Some of these classes inherit from other classes and implement interfaces. Classes further declare methods and attributes: the methods invoke other methods and expect zero or more arguments each; the attributes can be accessed by methods.

The following questions are related to the Java code of *Apache Ivy*. You may freely browse the code in the *Package Explorer* and *Java Editor* of Eclipse. You must neither use the *Eclipse Search* nor the features found in the context menus of Eclipse. Instead, you should make use of the Natural Language Query Interface.

Please respect the time limits given for each task.

7. All the (direct) subclasses of <*ivy*>/util/AbstractMessageLogger? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

8. All methods with <*ivy*>/osgi/updatesite/xml/EclipseFeature as argument (parameter)? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

9. All methods that invoke both,
<*ivy*>core/module/descriptor/DefaultModuleDescriptor/addDependency(DependencyDescriptor)
and
<*ivy*>osgi/core/BundleInfo/getRequirements()? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

10. Reflecting on the tasks you just solved, did you come up with particular questions that are not yet supported by the tool? If so, please write them down, together with the associated task number(s).

.....

.....

.....

.....

.....

Questions concerning the Development History

> Whenever a developer of *Apache Ivy* has changed a file in the past, he committed a new revision to the Subversion repository.

The following questions are related to the development history of *Apache Ivy*. You are **not** allowed to use the *Subversion* plug-in (history view, repository browsing perspective, etc.). Again, use the Natural Language Query Interface instead.

Please respect the time limits given for each task.

11. The developers of *Apache Ivy* that have changed <*ivy*>/plugins/lock/AbstractLockStrategy.java in the past? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

12. The file of *Apache Ivy* that has changed most often in the past? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

13. The last five files changed by Maarten Coene? (5min)

.....

.....

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

14. Reflecting on the tasks you just solved, did you come up with particular questions that are not yet supported by the tool? If so, please write them down, together with the associated task number(s).

.....

.....

.....

.....

.....

Questions concerning Defects

When a stakeholder finds an issue in *Apache Ivy*, she or he can report it online in *Jira*. Several categories of issues exists, such as *Bugs*, *Improvements*, *Feature Requests*, etc. Issues are then assigned to a developer for fixing them. Open issues are those that are unresolved and ready for the assignee to start work on them. Once they are being actively worked on, issues are considered 'in progress' until they are resolved. Resolution can become more difficult, if issues depend on each other. For example, when one issue blocks another, the first one needs to be fixed before the second can be attacked. Before an issue can finally be closed, the resolution needs to be verified by the reporter.

The following questions are related to issues of *Apache Ivy*. You are **not** allowed to use the Mylyn plug-in of Eclipse or to browse the issues online. Use the Natural Language Query Interface instead.

Please respect the time limits given for each task.

15. What feature requests were implemented by *Maarten Coene*? (5min)

.....

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**

Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

16. The issues that *Nicolas Lalevée* and *Xavier Hanin* both commented on? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

17. The issues blocked by Issue IVY-739? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

18. Reflecting on the tasks you just solved, did you come up with particular questions that are not yet supported by the tool? If so, please write them down, together with the associated task number(s).

.....

.....

.....

.....

.....

Cross-domain Questions

> Whenever *Apache Ivy*'s source code is changed in consequence to an issue report, developers commit these changes and add a reference to the issue to the commit message, *e.g.*, "FIX: Invalid error report with m2compatible resolver (IVY-456)."

The following questions span multiple domains: they are related to the code, the issues, and the development history. You may use the Eclipse *Package Explorer*, *Java Editor*, and the Natural Language Query Interface.

Please respect the time limits given for each task.

19. The classes that were affected by Issue IVY-764? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

20. The issues that affected <*ivy*>/plugins/resolver/IvyRepResolver? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

21. Most error-prone class in *Apache Ivy*? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

22. The issues that affected <*ivy*>/plugins/resolver/AbstractResolver and <*ivy*>/core/cache/DefaultRepositoryCacheManager? (5min)

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

23. Reflecting on the tasks you just solved, did you come up with particular questions that are not yet supported by the tool? If so, please write them down, together with the associated task number(s).

.....

.....

.....

.....

.....

Usability

The following questions are part of the **System Usability Scale** by Brooks, Digital Equipment Corporation, 1986. They are commonly used to evaluate the usability of software systems. Please provide answers in respect to the features of the Natural Language Query Interface.

All items should be checked. Please record your immediate response to each item, rather than thinking about items for a long time. If you feel that you cannot respond to a particular item, you should mark the centre point of the scale.

		I strongly **disagree**				I strongly **agree**
24a.	I think that I would like to use this system frequently	○	○	○	○	○
24b.	I found the system unnecessarily complex	○	○	○	○	○
24c.	I thought the system was easy to use	○	○	○	○	○
24d.	I think that I would need the support of a technical person to be able to use this system	○	○	○	○	○
24e.	I found the various functions in this system were well integrated	○	○	○	○	○
24f.	I thought there was too much inconsistency in this system	○	○	○	○	○
24g.	I would imagine that most people would learn to use this system very quickly	○	○	○	○	○
24h.	I found the system very cumbersome to use	○	○	○	○	○
24i.	I felt very confident using the system	○	○	○	○	○
24j.	I needed to learn a lot of things before I could get going with this system	○	○	○	○	○

Debriefing

Thank you for participating in our evaluation. To conclude our experiment, we would like to ask you for some feedback on our tool.

25. What did you like most about the Natural Language Interface?

26. What did you like least about the Natural Language Interface?

27. Do you have any comments and/or suggestions about the evaluation, that could help us to improve it?

28. Are there any additional comments that you would like to share with us?

Contact Information

Please write down your contact information if you are interested in the results of this experiment. We will treat your data confidential and not use it for any other purpose.

First Name:

Last Name:

Email Address:

Evaluation of Hawkshaw

Control Group – Eclipse Tools & Browser

Department of Informatics, University of Zurich

27. April 2012

Dear Participant

The goal of this experiment is to evaluate HAWKSHAW, a query system for software evolution data, in comparison to more traditional software engineering tools. We appreciate your time very much—your support in this experiment is invaluable for the outcome of our research.

In the following, you will be asked to answer a set of questions about *Apache Ivy*, a popular open source Java tool for managing project dependencies. The questions will be about its source code and development & defect history, and each question has one or several answers. You belong to the control group and will answer the questions with the tools that are part of Eclipse.

You are asked:

- not to consult any other participants throughout the course of the experiment.
- to use only the tools and features listed in the beginning of each section.
- to solve all the tasks in their given order by filling in the requested information.
- to solve the tasks as fast as possible. For each task with time restriction, the available time is given in parentheses. Do not use more than the amount of time assigned to each task. In case that you cannot finish a task within the allotted time, you have to proceed to the next one.
- to write down how much time you have spent on each of the tasks (including the time needed for writing down the answer) in the text boxes printed at the end of the tasks (even if you could not complete a task). Use the *Experiment Timer* view to keep track of time. For the questions concerned with your personal details in the next section, there are no time restrictions—neither are there any for the debriefing section at the end of the questionnaire.

- to quickly asses the practical relevance (*i.e.*, how realistic is the task?) and difficulty of each task by checking the appropriate checkbox. By "realistic", we mean how likely you think that you would be interested to answer such a question in your daily work as a developer.

For any questions where full sentences are required from you, you can use either English or German in your answers—what ever you are more comfortable with.

Thank you for participating in our experiment,
Michael Würsch and Colleagues

☞ Please read now the Eclipse tutorial: `doc/eclipse-tutorial.pdf`

Personal Details

We kindly ask you to share below some personal information about you. It is used exclusively for statistical purposes and will not be traced back to specific individuals.

About yourself

1. How old are you? ..... years.

2. What is your gender?

   ○ Female

   ○ Male

Your experience

3. What is your current occupation?

   ○ Bachelor's student

   ○ Master's student

   ○ PhD student

   ○ Practitioner

   ○ Other: .....

4. How many years of development expertise do you have (any language)? ..... years.

5. How many years of Java development expertise with Eclipse do you have? ..... years.

Please tells us about your skills in the following areas:

		n/a	poor	average	good	excellent
6a.	My English skills are	○	○	○	○	○
6b.	My programming skills are:	○	○	○	○	○
6c.	My Java skills are:	○	○	○	○	○
6d.	My familiarity w/ *Jira* is:	○	○	○	○	○
6e.	My familiarity w/ *Subversion* is:	○	○	○	○	○

☞Please start tracking time with the Experiment Timer when you begin with the next task

Questions concerning Program Code

Apache Ivy is an object-oriented system implemented in Java. It is comprised of approx. 50'000 lines of code in 64 packages and 668 classes. Some of these classes inherit from other classes and/or implement interfaces. Classes further declare methods and attributes: the methods invoke other methods and expect zero or more arguments each; the attributes can be accessed by methods.

The following questions are related to the Java code of *Apache Ivy*. You may freely browse the code in the *Package Explorer* and *Java Editor* of Eclipse. You can also make use of the *Eclipse Search* and the features found in the context menus of Eclipse. Do **not** use the Natural Language Query Interface.

Please respect the time limits given for each task.

7. All the (direct) subclasses of <*ivy*>/util/AbstractMessageLogger? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

8. All methods with <*ivy*>/osgi/updatesite/xml/EclipseFeature as argument (parameter)? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

9. All methods that invoke both,
<*ivy*>core/module/descriptor/DefaultModuleDescriptor/addDependency(DependencyDescriptor)
and
<*ivy*>osgi/core/BundleInfo/getRequirements()? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

Questions concerning the Development History

> Whenever a developer of *Apache Ivy* has changed a file in the past, he committed a new revision to the Subversion repository.

The following questions are related to the development history of *Apache Ivy*. You may use any Eclipse feature, **except** for our Natural Language Query Interface. You might want to make use of the functionality of the *Subversion* Eclipse plug-in, in particular its *History View*.

Please respect the time limits given for each task.

10. The developers of *Apache Ivy* that have changed <*ivy*>/plugins/lock/AbstractLockStrategy.java in the past? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

11. The file of *Apache Ivy* that has changed most often in the past? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

12. The last five files changed by Maarten Coene? (5min)

.....

.....

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

Questions concerning Defects

When a stakeholder finds an issue in *Apache Ivy*, she or he can report it online in *Jira*. Several categories of issues exists, such as *Bugs*, *Improvements*, *Feature Requests*, etc. Issues are then assigned to a developer for fixing them. Open issues are those that are unresolved and ready for the assignee to start work on them. Once they are being actively worked on, issues are considered 'in progress' until they are resolved. Resolution can become more difficult, if issues depend on each other. For example, when one issue blocks another, the first one needs to be fixed before the second can be attacked. Before an issue can finally be closed, the resolution needs to be verified by the reporter.

The following questions are related to issues of *Apache Ivy*. You may use any Eclipse feature, **except** for our Natural Language Query Interface. You can also browse *Apache Ivy*'s issue tracker `https://issues.apache.org/jira/browse/IVY` online.

Please respect the time limits given for each task.

13. What feature requests were implemented by *Maarten Coene*? (5min)

.....

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

14. The issues that *Nicolas Lalevée* and *Xavier Hanin* both commented on? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

15. The issues blocked by Issue IVY-739? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

Cross-domain Questions

> Whenever *Apache Ivy*'s source code is changed in consequence to an issue report, developers commit these changes and add a reference to the issue to the commit message, *e.g.*, "FIX: Invalid error report with m2compatible resolver (IVY-456)."

The following questions span multiple domains: they are related to the code, the issues, and the development history. You may use any Eclipse feature, **except** for our Natural Language Query Interface. You can also browse *Apache Ivy*'s issue tracker `https://issues.apache.org/jira/browse/IVY` online.

Please respect the time limits given for each task.

16. The classes that were affected by Issue IVY-764? (5min)

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

17. The issues that affected <*ivy*>/plugins/resolver/IvyRepResolver? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

18. Most error-prone class in *Apache Ivy*? (5min)

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

19. The issues that affected <*ivy*>/plugins/resolver/AbstractResolver and <*ivy*>/core/cache/DefaultRepositoryCacheManager? (5min)

.....

.....

.....

Overall, this task was?	Very **unrealistic**	○	○	○	○	○	○	○	Very **realistic**
Overall, this task was?	Very **Difficult**	○	○	○	○	○	○	○	Very **Easy**

Time spent on this task:

: *mm:ss*

Usability

The following questions are part of the **System Usability Scale** by Brooks, Digital Equipment Corporation, 1986. They are commonly used to evaluate the usability of software systems. Please provide answers in respect to all the features of the Eclipse IDE that you have used in this experiment.

All items should be checked. Please record your immediate response to each item, rather than thinking about items for a long time. If you feel that you cannot respond to a particular item, you should mark the centre point of the scale.

		I strongly **disagree**				I strongly **agree**
20a.	I think that I would like to use this system frequently	○	○	○	○	○
20b.	I found the system unnecessarily complex	○	○	○	○	○
20c.	I thought the system was easy to use	○	○	○	○	○
20d.	I think that I would need the support of a technical person to be able to use this system	○	○	○	○	○
20e.	I found the various functions in this system were well integrated	○	○	○	○	○
20f.	I thought there was too much inconsistency in this system	○	○	○	○	○
20g.	I would imagine that most people would learn to use this system very quickly	○	○	○	○	○
20h.	I found the system very cumbersome to use	○	○	○	○	○
20i.	I felt very confident using the system	○	○	○	○	○
20j.	I needed to learn a lot of things before I could get going with this system	○	○	○	○	○

Debriefing

Thank you for participating in our evaluation. To conclude our experiment, we would like to ask you for some feedback on the Eclipse tools, you have used.

21. What functionality of Eclipse or *Jira* was most useful to you in solving the tasks?

22. Were you missing any functionality or was a feature particularly difficult to use?

23. Do you have any comments and/or suggestions about the evaluation, that could help us to improve it?

24. Are there any additional comments that you would like to share with us?

Contact Information

Please write down your contact information if you are interested in the results of this experiment. We will treat your data confidential and not use it for any other purpose.

First Name:

Last Name:

Email Address:

Bibliography

[AB74] George W. Adamson and Jillian Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974. Cited on page 124.

[AOH07] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007. Cited on page 147.

[Ask94] Ulf Asklund. Identifying conflicts during structural merge. In *Proceedings of the Nordic Workshop Programming Environment Research*, pages 231–242, 1994. Cited on page 149.

[AZ11] Awny Alnusair and Tian Zhao. Retrieving reusable software components using enhanced representation of domain knowledge. In *Recent Trends in Information Reuse and Integration*, pages 363–379. Springer, Wien, Austria, 2011. Cited on page 75.

[BAJ11] Hamid Abdul Basit, Usman Ali, and Stan Jarzabek. Viewing simple clones from structural clones' perspective. In *Proceedings of the 5th International Workshop on Software Clones*, pages 1–6, 2011. Cited on page 181.

[BB10] Cosmin Basca and Abraham Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *Proceedings of the 6th International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 64–79, 2010. Cited on page 17.

[BCH10] Chris Bizer, Richard Cyganiak, and Tom Heath. How to publish linked data on the web. `http://www4.wiwiss.fu-berlin.de/bizer/pub/LinkedDataTutorial/`, Last visited Jan. 2010. Cited on page 44.

[BE96] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996. Cited on page 1.

[BEJWKG05] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim, and Michael W. Godfrey. Facilitating software evolution research with Kenyon. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, 2005. Cited on pages 36, 50, and 85.

[Bil05] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1–3):217–239, Jun 2005. Cited on page 146.

[BKK05] Abraham Bernstein, Christoph Kiefer, and Esther Kaufmann. SimPack: A generic java library for similarity measures in ontologies. Technical report, Department of Informatics, University of Zurich, Switzerland, 2005. Cited on page 123.

[BKKK06] Abraham Bernstein, Esther Kaufmann, Christian Kaiser, and Christoph Kiefer. Ginseng: A guided input natural language search engine for querying ontologies. In *Jena User Conference*, pages 2–4, 2006. Cited on pages 11, 20, 68, 94, 96, and 162.

[BKM08] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008. Cited on pages 176 and 189.

[BKPS97] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk. In *Workshop on Modeling and Empirical Studies of Software Engineering*, 1997. Cited on page 55.

[BLFM98] Tim Berners-Lee, R. Fielding, and L. Masinter. RFC 2396 - uniform resource identifiers (URI). IETF RFC, August 1998. `http://www.ietf.org/rfc/rfc2396.txt`. Cited on pages 37, 48, 84, and 158.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific America*, 284(5):34–43, 2001. Cited on pages 48, 83, and 158.

[BOL09] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, 2009. Cited on pages 15 and 39.

[Bro87] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987. Cited on page 5.

[Bro95] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. Cited on pages 1 and 156.

[Bro96] John Brooke. SUS - a quick and dirty usability scale. In Patrick W. Jordan, editor, *Usability Evaluation in Industry*, pages 189–194. Taylor & Francis, 1996. Cited on pages 10, 25, 176, and 185.

[BRZ+10] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 455–464, 2010. Cited on page 26.

[BVG06] Manuel F. Bertoa, Antonio Vallecillo, and Félix Garcia. An ontology for software measurement. In *Ontologies for Software Engineering and Software Technology*, pages 175–196. Springer, Heidelberg, Germany, 2006. Cited on pages 56, 73, and 106.

[BYdM+98] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377, 1998. Cited on pages 127 and 149.

[CCP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 14–14, 2007. Cited on page 147.

[CGK98] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering*, 24(9):682–694, 1998. Cited on pages 38 and 50.

[Cho04] Gobinda G. Chowdhury. *Introduction to Modern Information Retrieval*. Facet, London, UK, 2nd edition, 2004. Cited on pages 6, 82, and 156.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992. Cited on page 105.

[CMSB05] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, Jun 2005. Cited on page 110.

[Cre97] Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997. Cited on pages 105 and 211.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996. Cited on pages 23, 109, 111, 112, 113, 120, and 130.

[dAM08] Brian de Alwis and Gail C. Murphy. Answering conceptual queries with ferret. In *Proceedings of the 30th International Conference on Software Engineering*, pages 21–30, 2008. Cited on pages 4, 21, 82, 97, 98, 104, 166, 173, 175, and 212.

[DaVAdLM08] Frederico A. Durão, Taciana A. Vanderlei, Eduardo S. Almeida, and Silvio R. de L. Meira. Applying a semantic layer in a source code search tool. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1151–1157, 2008. Cited on page 76.

[DBS91] Prem Devanbu, Ron Brachman, and Peter G. Selfridge. LaSSIE: a knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991. Cited on pages 30, 74, 104, and 211.

[De04] Mike Dean and Guus Schreiber eds. *OWL Web Ontology Language Reference*. W3C Recommendation, February 2004. `http://www.w3.org/TR/owl-ref/`. Cited on pages 14, 39, 49, 84, 158, and 159.

[DE05] Jens Dietrich and Chris Elgar. A formal description of design patterns using OWL. In *Proceedings of the Australian Software Engineering Conference*, pages 243–250, 2005. Cited on page 75.

[DGLP08] Marco D'Ambros, Harald C. Gall, Michele Lanza, and Martin Pinzger. Analyzing software repositories to understand software evolution. In *Software Evolution*, pages 37–67. Springer, Heidelberg, Germany, 2008. Cited on pages 36 and 49.

[Dic45] Lee R. Dice. Measures of the amount of ecologic association between species. *ESA Ecology*, (26):297–302, 1945. Cited on pages 124 and 149.

[DLP07] Marco D'Ambros, Michele Lanza, and Martin Pinzger. "a bug's life" – visualizing a bug database. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 113–120, 2007. Cited on page 2.

[DLR12] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive

comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012. Cited on page 2.

[DTS01] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.1 – the FAMOOS information exchange model. Technical report, University of Bern, 2001. Cited on page 149.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000. Cited on pages 1 and 26.

[FBCC+10] David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An overview of the DeepQA project. *AI Magazine*, 31(3):59–79, 2010. Cited on page 28.

[Fer11] David A. Ferrucci. IBM's Watson/DeepQA. In *Proceedings of the 38th annual international symposium on Computer architecture*, 2011. Cited on pages 28 and 207.

[FG06] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, 2006. Cited on pages 24, 110, 133, and 134.

[Fie00] Roy Thomas Fielding. *Architectural Styles and Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. Cited on page 66.

[FL07] Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema. W3C Recommendation, 28 August 2007. `http://www.w3.org/TR/sawsdl/`. Cited on page 67.

[FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184, 2010. Cited on pages 173, 174, 195, 198, and 212.

[FPG03a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 90, 2003. Cited on page 22.

[FPG03b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, 2003. Cited on pages 2, 3, 22, 36, 50, and 85.

[FWPG07] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change Distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007. Cited on pages 55, 86, and 169.

[GDL04] Tudor Girba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, 2004. Cited on page 2.

[GFP09] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, 2009. Cited on pages 15, 36, 39, 50, and 86.

[GG11] Giacomo Ghezzi and Harald C. Gall. SOFAS: A lightweight architecture for software analysis as a service. In *Proceedings of the 9th IEEE/IFIP Working Conference on Software Architecture*, pages 93–102, 2011. Cited on pages 27, 66, 67, and 73.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, page 190, 1998. Cited on pages 2, 55, and 110.

[GL02] Michael Gruninger and Jintae Lee. Ontology applications and design. *Communications of the ACM*, 45(2):39–41, 2002. Cited on pages 73 and 105.

[GPG10] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56, 2010. Cited on page 58.

[Gru93] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. Cited on pages 37, 48, 83, and 158.

[GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, Feb 2005. Cited on page 148.

[Had09] Marc Hadley. Web application description language (wadl). W3C Member Submission, 31 August 2009. `http://www.w3.org/Submission/wadl/`. Cited on page 67.

[HDLL11] Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. Software evolution comprehension: Replay to the rescue. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, pages 161–170, 2011. Cited on pages 173 and 174.

[Hen94] Scott Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11(5):48–59, 1994. Cited on pages 6, 82, and 156.

[HGWG11] Matthias Hert, Giacomo Ghezzi, Michael Würsch, and Harald C. Gall. How to "make a bridge to the new town" using OntoAccess. In *Proceedings of the 10th International Semantic Web Conference*, pages 112–127, 2011. Cited on page 30.

[HH04] Ahmend E. Hassan and Richard C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 76–81, 2004. Cited on page 146.

[HKF08] Olaf Hartig, Martin Kost, and Johann-Christoph Freytag. Automatic component selection with semantic technologies. In *Proceedings of the 4th International Workshop on Semantic Web Enabled*

Software Engineering, held at the 7th International Semantic Web Conference, 2008. Cited on page 75.

[HKST06] Hans-Jörg Happel, Axel Korthaus, Stefan Seedorf, and Peter Tomczyk. KOntoR: An ontology-enabled approach to software reuse. *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, pages 349–354, 2006. Cited on pages 75 and 106.

[Hol79] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. Cited on page 191.

[Hor90] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990. Cited on page 147.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission, May 2004. `http://www.w3.org/Submission/SWRL/`. Cited on pages 64 and 74.

[HPVS09] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st IEEE International Conference on Software Engineering*, pages 232–242, 2009. Cited on pages 104 and 211.

[HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977. Cited on page 146.

[HS04] Susan J Hespos and Elizabeth S Spelke. Conceptual precursors to language. *Nature*, 430(6998):453–456, 2004. Cited on page 55.

[HS06] Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proceedings of the 2nd International*

Workshop on Semantic Web Enabled Software Engineering, held at the 5th International Semantic Web Conference, 2006. Cited on pages 73 and 105.

[HSP07] Catalina Hallett, Donia Scott, and Richard Power. Composing questions through conceptual authoring. *Computational Linguistics*, 33(1):105–133, 2007. Cited on pages 6, 68, and 160.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley, Boston, MA, USA, 1999. Cited on page 62.

[Hun01] James J. Hunt. *Extensible, Language-Aware Differencing and Merging*. PhD thesis, University of Karlsruhe, 2001. Cited on page 149.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with datalog. In *ECOOP 2006 – Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2006. Cited on pages 5, 82, 105, 156, and 211.

[HWCK06] David Hyland-Wood, David Carrington, and Simon Kaplan. Toward a software maintenance methodology using semantic web techniques. In *Proceedings of the 2nd IEEE International Workshop on Software Evolvability*, pages 23–30, 2006. Cited on pages 73 and 105.

[IUHT09] Aftab Iqbal, Oana Ureche, Michael Hausenblas, and Giovanni Tummarello. LD2SD: Linked data driven software development. In *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering*, pages 240–245, 2009. Cited on page 77.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, Feb 1912. Cited on page 124.

[JL94] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, pages 243–252, 1994. Cited on page 147.

[JV03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003. Cited on pages 5, 82, 105, 156, and 211.

[KB07] Esther Kaufmann and Abraham Bernstein. How Useful are Natural Language Interfaces to the Semantic Web for Casual End-users? In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference*, pages 281–294, 2007. Cited on page 97.

[KB10] Esther Kaufmann and Abraham Bernstein. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):377–393, 2010. Cited on page 212.

[KBL08] Christoph Kiefer, Abraham Bernstein, and André Locher. Adding data mining support to SPARQL via statistical relational learning methods. In *Proceedings of the 5th European Semantic Web Conference*, pages 478–492, 2008. Cited on page 77.

[KBS07] Christoph Kiefer, Abraham Bernstein, and Markus Stocker. The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic web tasks. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference*, page 295, 2007. Cited on page 77.

[KBT07] Christoph Kiefer, Abraham Bernstein, and Jonas Tappolet. Mining software repositories with iSPAROL and a software evolution ontology. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 10, 2007. Cited on pages 43, 77, and 106.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19:77–131, 2007. Cited on pages 36 and 50.

[KDV07] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, pages 344–353, 2007. Cited on pages 4, 173, and 174.

[Ke04] Graham Klyne and Jeremy J. Carroll eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, February 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`. Cited on pages 14, 37, 48, 84, and 158.

[KMM11] Markus Kimmig, Martin Monperrus, and Mira Mezini. Querying source code with natural language. In *Proceedings of the 26th IEEE/ACM International Conference On Automated Software Engineering*, pages 376–379, 2011. Cited on page 211.

[KN06] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 58–64, 2006. Cited on page 146.

[KNG07] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, 2007. Cited on page 148.

[KPEJW05] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, 2005. Cited on page 148.

[KRSR10] Iman Keivanloo, Laleh Roostapour, Philipp Schugerl, and Juergen Rilling. Semantic Web-based source code search. In *Proceedings of the 6th International Workshop on Semantic Web Enabled Software Engineering*, 2010. Cited on page 76.

[KSG+10] Ilya Kupershmidt, Qiaojuan J. Su, Anoop Grewal, Suman Sundaresh, Inbal Halperin, James Flynn, Mamatha Shekar, Helen Wang, Jenny Park, Wenwu Cui, Gregory D. Wall, Robert

Wisotzkey, Satnam Alag, Saeid Akhtari, and Mostafa Ronaghi. Ontology-based meta-analysis of global collections of high-throughput public data. *PLoS ONE*, 5(9), 2010. Cited on pages 28, 46, and 207.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196, 2005. Cited on page 62.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for uml models. In *Software Engineering*, pages 105–116, 2005. Cited on page 148.

[KZWJZ07] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007. Cited on pages 174 and 201.

[Leh79] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1(0):213–221, 1979. Cited on page 1.

[Leh80] Many M. Lehman. Programs, life cycles and laws of software evolution. In *Proceedings of the IEEE*, pages 1060–1076, 1980. Cited on pages 2 and 110.

[Lev66] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, Feb 1966. Cited on page 123.

[Lew92] James R. Lewis. Psychometric evaluation of the post-study system usability questionnaire: The PSSUQ. In *Proceedings of the Human Factors Society*, pages 1259–1263, 1992. Cited on page 176.

[LMD05] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer, Heidelberg, Germany, 2005. Cited on page 56.

[LR96] Christopher M. Lott and H. Dieter Rombach. Repeatable software engineering experiments for comparing defect-detection techniques. *Empirical Software Engineering*, 1(3):241–277, 1996. Cited on page 204.

[LS09] James R. Lewis and Jeff Sauro. The factor structure of the system usability scale. In *Proceedings of the 1st International Conference on Human Centered Design*, pages 94–103, 2009. Cited on pages 185 and 189.

[LTP04] Timothy C. Lethbridge, Sander Tichelaar, and Erhard Plödereder. The Dagstuhl Middle Metamodel: a schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, (94):7–18, May 2004. Cited on pages 14 and 50.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, 2006. Cited on pages 1, 4, 5, 25, 173, and 174.

[MC04] Jonathan I. Maletic and Michael L. Collard. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 210–219, 2004. Cited on page 147.

[Men02] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002. Cited on page 149.

[MFGW12] Sebastian Müller, Thomas Fritz, Harald Gall, and Michael Würsch. An approach for collaborative code reviews using multi-touch technology. In *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2012. Cited on page 27.

[Mil95] George A. Miller. WordNet: a lexical database for english. *Communications of the ACM*, 38:39–41, 1995. Cited on pages 29 and 209.

[MK88] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988. Cited on page 50.

[mos] MOST (Marrying Ontology and Software Technology) Project homepage. `http://www.most-project.eu/`. Cited on page 106.

[Nor10] Marija Norusis. *SPSS 18 Advanced Statistical Procedures Companion*. Prentice Hall, New Jersey, NY, USA, 2010. Cited on page 189.

[Obj98] Object Management Group. XML Metadata Interchange (XMI). Technical Report OMG Document ad/98-10-05, February 1998. Cited on pages 38 and 50.

[OGS09] Daniel Oberle, Stephan Grimm, and Steffen Staab. An ontology for software. In *Handbook on Ontologies in Information Systems*, pages 392–412. Springer, Heidelberg, Germany, 2nd edition, 2009. Cited on page 73.

[Pe08] Eric Prud'hommeaux and Andy Seaborne eds. SPARQL query language for RDF. W3C Recommendation, January 2008. `http://www.w3.org/TR/rdf-sparql-query/`. Cited on pages 15, 37, 48, 84, and 158.

[PFD11] Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371, 2011. Cited on page 181.

[PGG07] Martin Pinzger, Emanuel Giger, and Harald C. Gall. Handling unresolved method bindings in eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007. Cited on pages 20 and 89.

[PGKG08] Martin Pinzger, Katja Gräfenhain, Patrick Knab, and Harald C. Gall. A tool for visual understanding of source code dependencies. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 254–259, 2008. Cited on page 86.

[PP96] Santanu Paul and Ataul Prakash. A query algebra for program databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996. Cited on pages 105 and 211.

[PP05] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, Jun 2005. Cited on page 148.

[PSE98] Richard Power, Donia Scott, and Roger Evans. What you see is what you meant: Direct knowledge editing with natural language feedback. In *Proceedings of the 13th Biennial European Conference on Artificial Intelligence*, pages 675–681, 1998. Cited on pages 68 and 160.

[PSHe04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks eds. *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/owl-semantics/`. Cited on pages 49 and 84.

[QL08] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. In *Proceedings of the 5th European Semantic Web Conference*, pages 524–538, 2008. Cited on page 17.

[RM07] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):Article No. 3, 2007. Cited on page 30.

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–70, 1975. Cited on page 2.

[RRL+04] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code base. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, 2004. Cited on page 147.

[Sau10] Jeff Sauro. If you could only ask one question, use this one. Personal Blog, March 2010. `http://www.measuringusability.com/blog/single-question.php`. Cited on page 177.

[Sau11] Jeff Sauro. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. CreateSpace Independent Publishing Platform, 2011. Cited on page 188.

[SBPK06] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 65–71, 2006. Cited on page 149.

[SD09] Jeff Sauro and Joseph S. Dumas. Comparison of three one-question, post-task usability questionnaires. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 1599–1608, 2009. Cited on page 177.

[SHPVS08] Giriprasad Sridhara, Emily Hill, Lori Pollock, and K Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 123–132, 2008. Cited on page 210.

[SL09] Jeff Sauro and James R. Lewis. Correlations among prototypical usability metrics: evidence for the construct of usability. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 1609–1618, 2009. Cited on page 205.

[SMDV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 23–34, 2006. Cited on pages 5, 21, 68, 82, 102, 160, and 173.

[SMH08] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A highly-efficient OWL reasoner. In *Proceedings of the 5th OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference*, 2008. Cited on page 49.

[SMV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008. Cited on pages 5, 21, 25, 82, 102, 173, 174, and 194.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal Web Semantics*, 5(2):51–53, 2007. Cited on pages 49, 70, 159, and 171.

[SSM05] Jelber Sayyad Shirabad and Tim J. Menzies. The PROMISE Repository of Software Engineering Databases., 2005. `http://promise.site.uottawa.ca/SERepository`. Cited on page 15.

[SZ90] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *Journal of Algorithms*, 11(4):581–621, Dec 1990. Cited on page 146.

[SZZ05a] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: Raising risk awareness. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107–110, 2005. Cited on pages 111 and 148.

[SZZ05b] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005. Cited on pages 22, 36, 50, and 148.

[Tai79] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the Association for Computing Machinery*, 26(3):422–433, 1979. Cited on page 146.

[Tap08] Jonas Tappolet. Semantics-aware software project repositories. In *Proceedings of the European Semantic Web Conference, Ph.D. Symposium*, 2008. Cited on pages 14, 76, and 170.

[Tap11] Jonas Tappolet. *Managing Temporal Graph Data While Preserving Semantics*. PhD thesis, University of Zurich, 2011. Cited on pages 30 and 210.

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX and XMI. In *Proceedings of the 7th Working Conference on Reverse Engineering*, page 296, 2000. Cited on pages 14, 20, 38, 50, 87, and 88.

[TG02] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 127–136, 2002. Cited on page 148.

[TPT05] Craig W. Thompson, Paul Pazandak, and Harry R. Tennant. Talk to your semantic web. *IEEE Internet Computing*, 9(6):75–78, 2005. Cited on page 162.

[UJ96] Mike Uschold and Robert Jasper. A framework for understanding and classifying ontology applications. In *Proceedings of the Workshop on Ontologies and Problem-Solving Methods*, pages 16–21, 1996. Cited on pages 73 and 105.

[Val02] Gabriel Valiente. *Algorithms on trees and graphs*. Springer, Heidelberg, Germany, 2002. Cited on page 149.

[WÖ6] Michael Würsch. Improving ChangeDistiller–improving abstract syntax tree based source code change detection. Master's thesis, University of Zurich, 2006. Cited on page 138.

[Wel97] Christopher A. Welty. Augmenting abstract syntax trees for program understanding. In *Proceedings of the 12th International Conference on Automated Software Engineering*, page 126, 1997. Cited on page 74.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 68–79, 1991. Cited on page 149.

[WG98] Johannes Weidl and Harald C. Gall. Binding object models to source code: An approach to object-oriented re-architecting. In *Proceedings of the 22nd International Computer Software and Applications Conference*, pages 26–31, 1998. Cited on page 124.

[WGH+12] Michael Würsch, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall. Seon - a pyramid of ontologies for software evolution and its applications. *Computing*, 94(11):857–885, 2012. Cited on page 167.

[WGRG10] Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 165–174, 2010. Cited on pages 68, 73, 156, 160, and 166.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the GXL graph exchange language. In *Revised Lectures on Software Visualization*, pages 324–336, Heidelberg, Germany, 2002. Springer. Cited on pages 38 and 50.

[WRDG10] Michael Würsch, Gerald Reif, Serge Demeyer, and Harald C. Gall. Fostering synergies: how semantic web technology could influence software repositories. In *Proceedings of the 2nd International ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, pages 45–48, 2010. Cited on page 49.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. Cited on page 172.

[WZR07] René Witte, Yonggang Zhang, and Juergen Rilling. Empowering software maintainers with semantic web technologies. In *Proceedings of the 4th European Semantic Web Conference*, pages 37–52, 2007. Cited on pages 74 and 106.

[XS05a] Zhenchang Xing and Eleni Stoulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, Oct 2005. Cited on page 148.

[XS05b] Zhenchang Xing and Eleni Stoulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005. Cited on pages 124 and 148.

[Yan91] Wuu Yang. Identifying syntactic differences between two programs. *Journal Software–Practice and Experience*, 21(7):739–755, 1991. Cited on page 147.

[Yan94] Wuu Yang. How to merge program texts. *Journal of Systems and Software*, 27(2):129–135, Nov 1994. Cited on page 149.

[YMNCC04] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004. Cited on page 110.

[YZY+08] Lian Yu, Jun Zhou, Yue Yi, Ping Li, and Qianxiang Wang. Ontology model-based static analysis on java programs. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pages 92–99, 2008. Cited on page 74.

[Zeh11] Stefan Zehnder. OntoX: a scriptable visualization framework for the semantic web. Master's thesis, 2011. Cited on page 31.

[Zha95] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 28(3):463–474, Jun 1995. Cited on page 146.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 2–6, 2004. Cited on page 85.

[ZWDZ05] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005. Cited on pages 85 and 110.

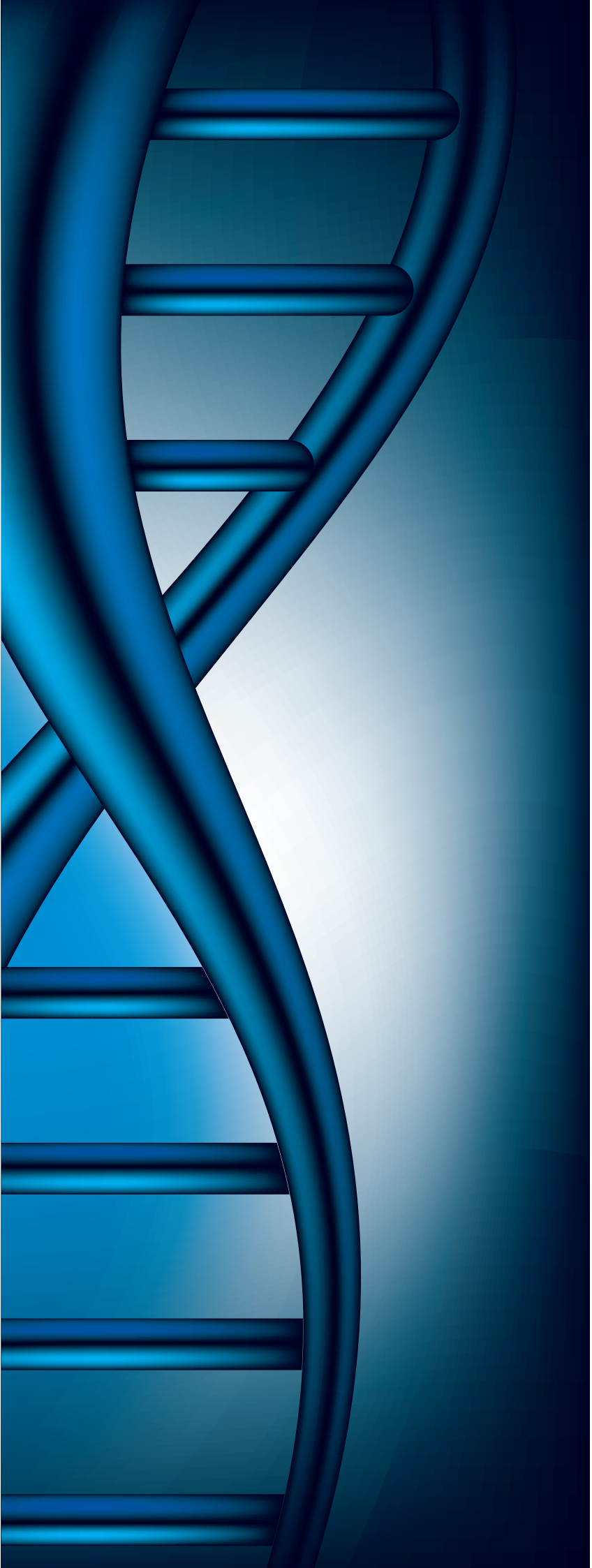
Curriculum Vitae

Personal Details

Name	Michael Würsch
Date of Birth	October 3, 1980
Place of Birth	Zurich, Switzerland
Citizenship	Swiss

Education

2012	*Doctor of Science UZH (summa cum laude)*
2007 – 2012	Doctoral Studies in Informatics Department of Informatics University of Zurich, Switzerland
2007	*Master of Science UZH (magna cum laude)*
2001 – 2007	Studies in Informatics Department of Informatics University of Zurich, Switzerland
2001	*Maturität Typus E*
1996 – 2001	Kantonsschule Hottingen (High School) Zurich, Switzerland
1993 – 1996	Sekundarschule (Middle School) Herrliberg, Switzerland
1987 – 1993	Primarschule (Elementary School) Herrliberg, Switzerland



A Query Framework for Software Evolution Data

The feature list of modern integrated development environments is steadily growing and mastering these tools becomes more and more demanding, especially for novice programmers.

Despite their remarkable capabilities, development environments often still cannot directly answer the questions that arise during program maintenance tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose.

We present a framework to query for information about a software system using a quasi-natural language interface that requires almost zero learning effort. Our approach is tightly woven into the Eclipse development environment and allows developers to answer questions related to source code, development history, or bug and issue management. For that, we model data extracted from various software repositories by means of ontologies, store them in a knowledge base of software evolution facts, and use knowledge processing techniques from the Semantic Web to query the knowledge base.

Our approach was evaluated in a user study with 35 subjects, who had to solve various software evolution tasks for an industrial-scale, open-source software system. The results of our user study showed that our query interface can outperform classical software engineering tools in terms of correctness, while yielding significant time savings to its users and greatly advancing the state of the art in terms of usability and learnability.

s.e.a.l.
software evolution & architecture lab